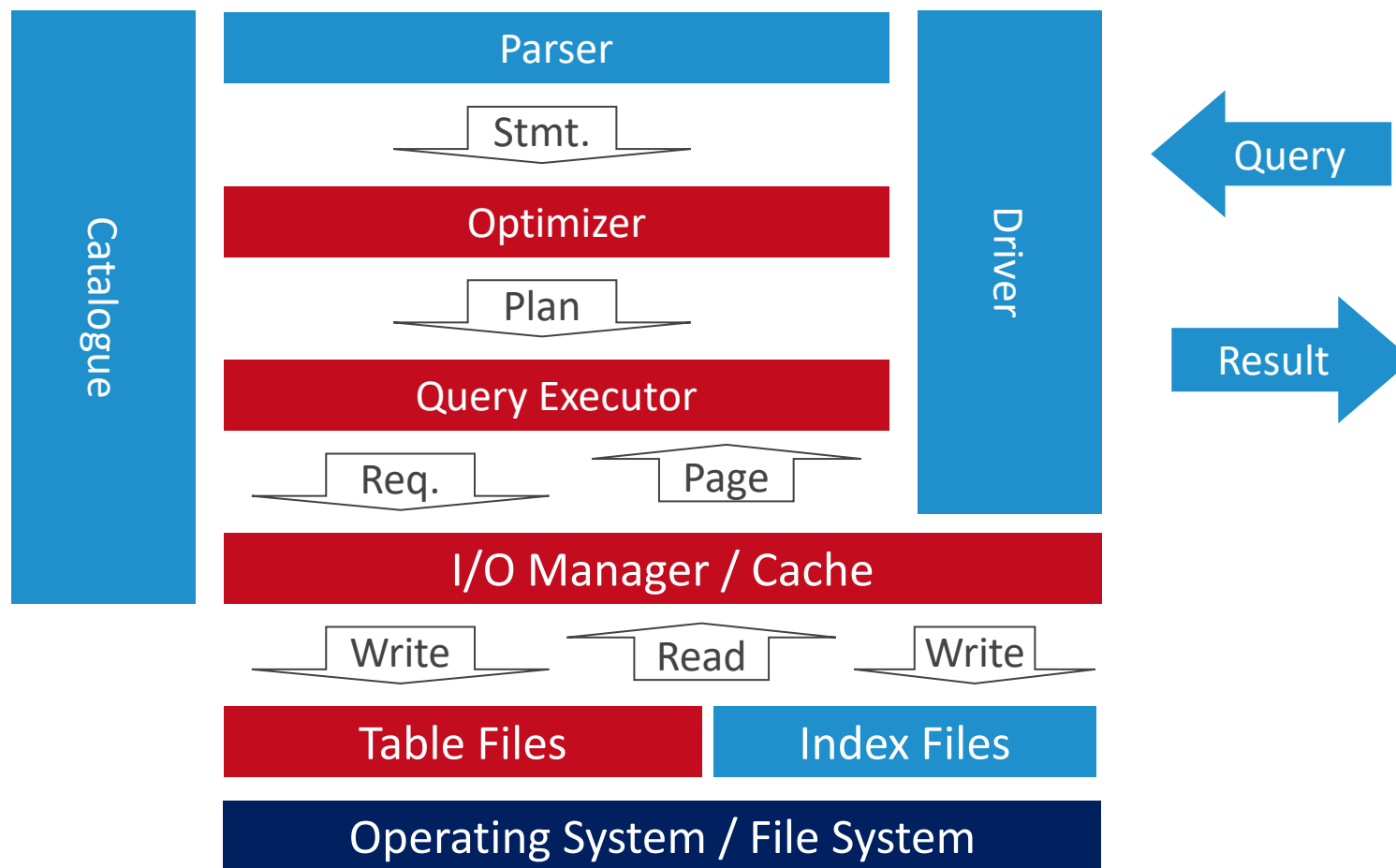


DBTLAB – Exercise 3: Buffer Pool Manager - I/O System

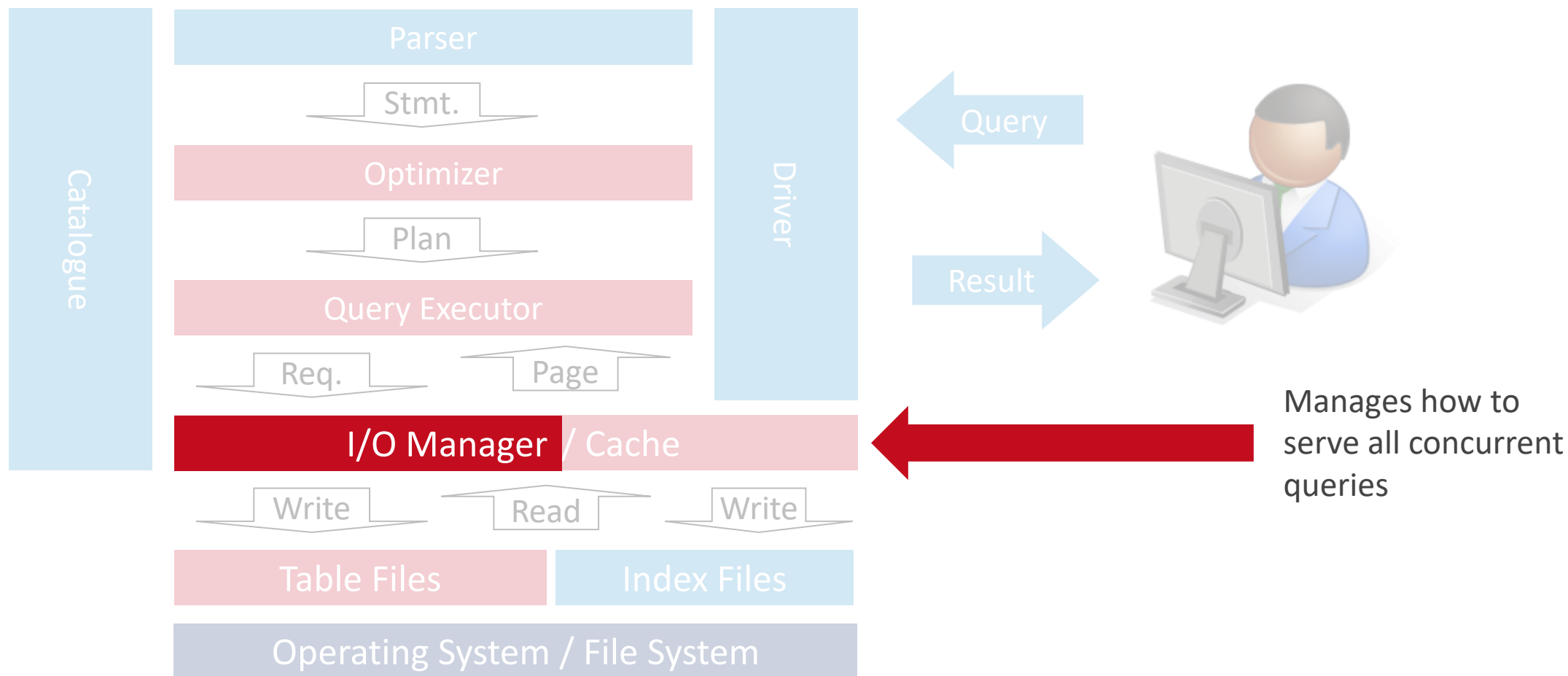
Rudi Poepel Lemaitre

based on material from Volker Markl, Sebastian Breß, Martin Kiefer

Basic system architecture

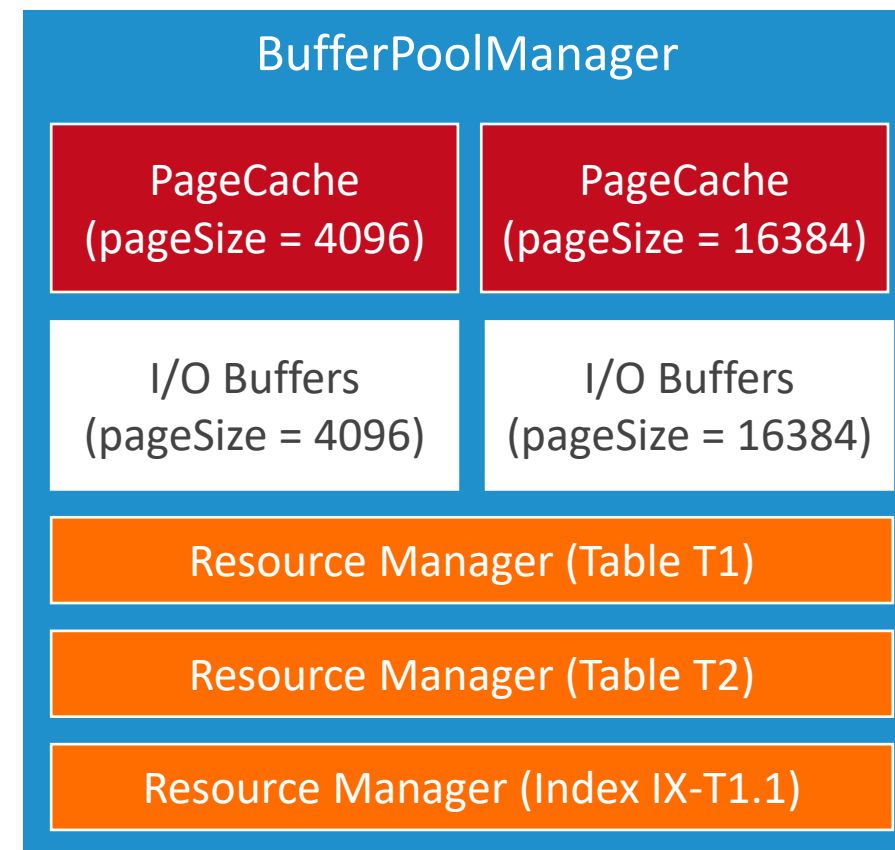


Where are we today?



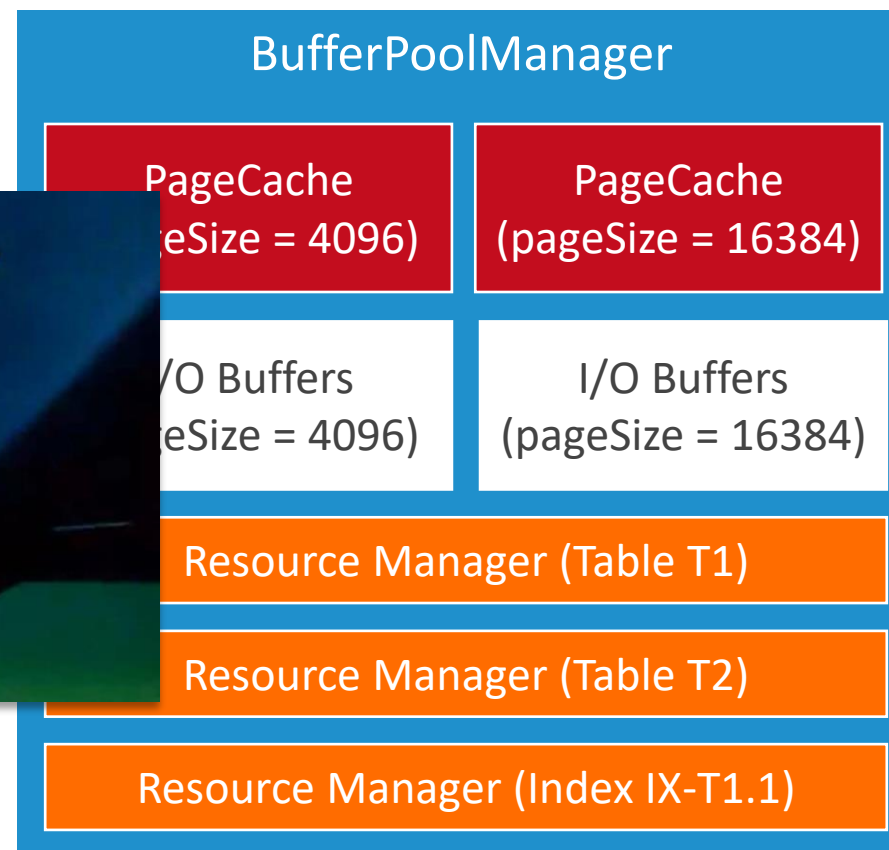
Buffer Pool Manager

- All pages are accessed through the *Buffer Pool Manager*.
- Manages multiple caches (one for each page size).
- Contains I/O buffers for reading / writing pages.
- Manages multiple resource managers (one for each table and index).
- Knows which resource (specific table or index) has its pages in which cache (depending on the page size).
- Checks the cache when a page is requested.
- Loads pages that were not in the cache.
- Resources are identified by a numeric resource ID.
 - Do not confuse with RID (row ID).
- The page size depends on the resource.



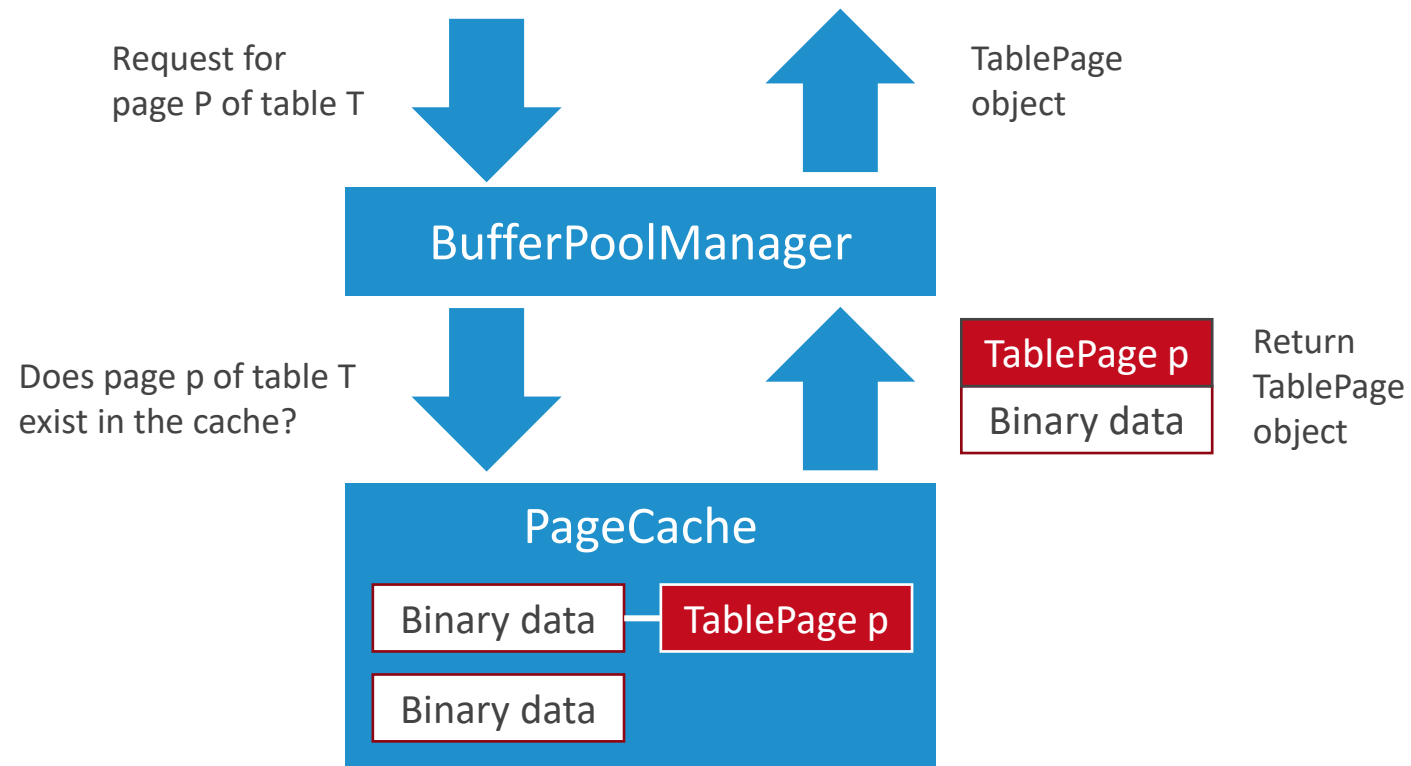
Buffer Pool Manager

- All pages are accessed through the *Buffer Pool Manager*.
- Manages multiple caches (one for each page size).
- Contains I/O buffers for reading / writing pages.
- Manages multiple resource managers.
- Knows which resource (specific table or index) is in which cache (depending on the page size).
- Checks the cache when a page is requested.
- Loads pages that were not in the cache.
- Resources are identified by a numeric resource ID.
 - Do not confuse with RID (row ID).
- The page size depends on the resource.



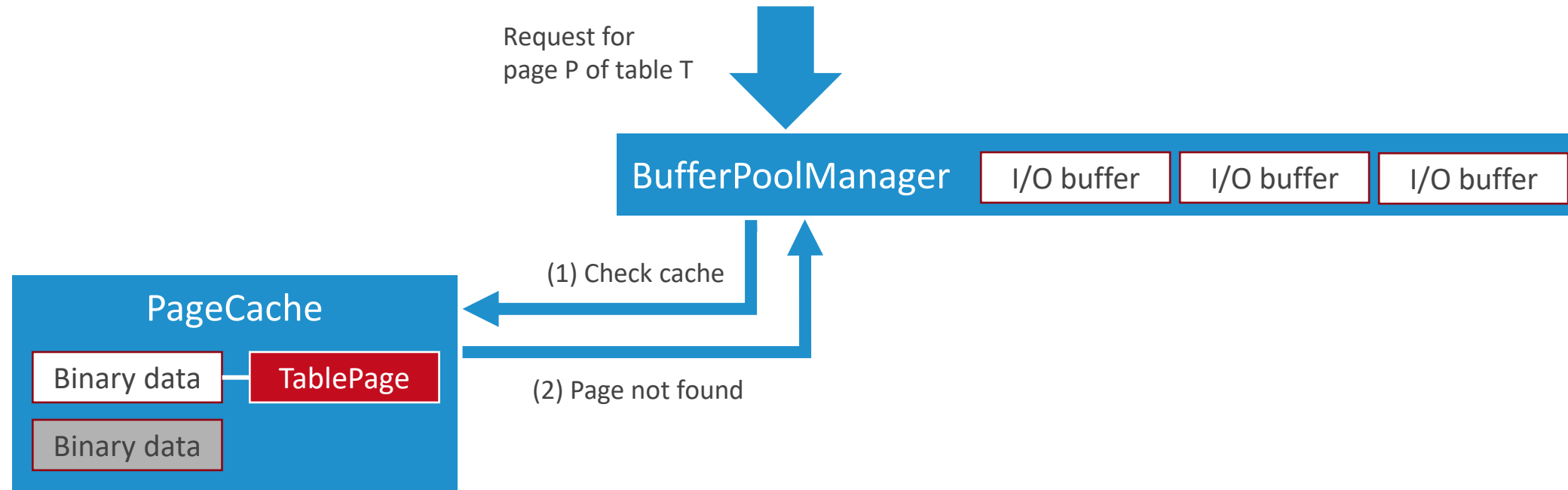
Recap: Retrieval of pages from the cache

- Case 1: Page exists in the cache (cache hit).



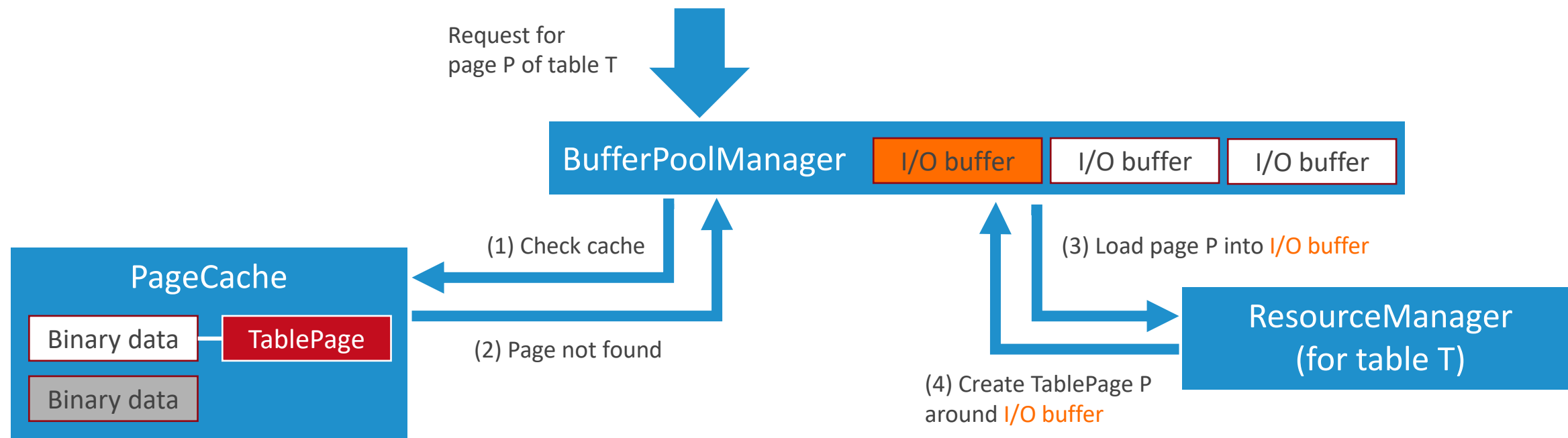
Recap: Retrieval of pages from disk

- Case 2: Page does not exist in the cache (cache miss).



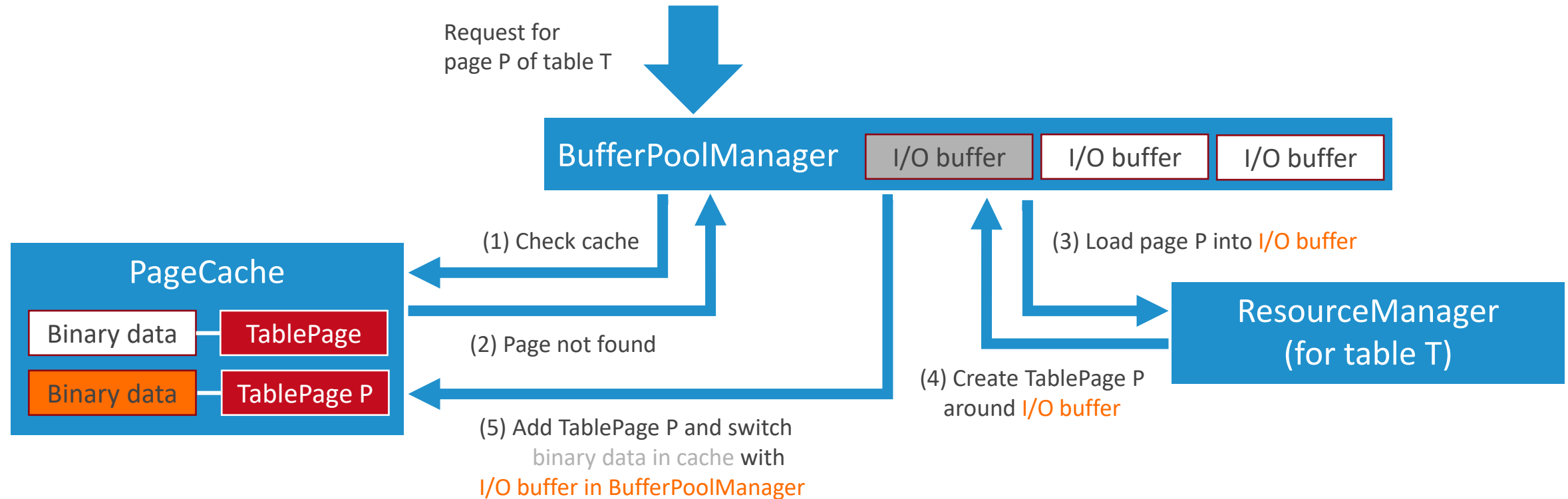
Recap: Retrieval of pages from disk

- Case 2: Page does not exist in the cache (cache miss).



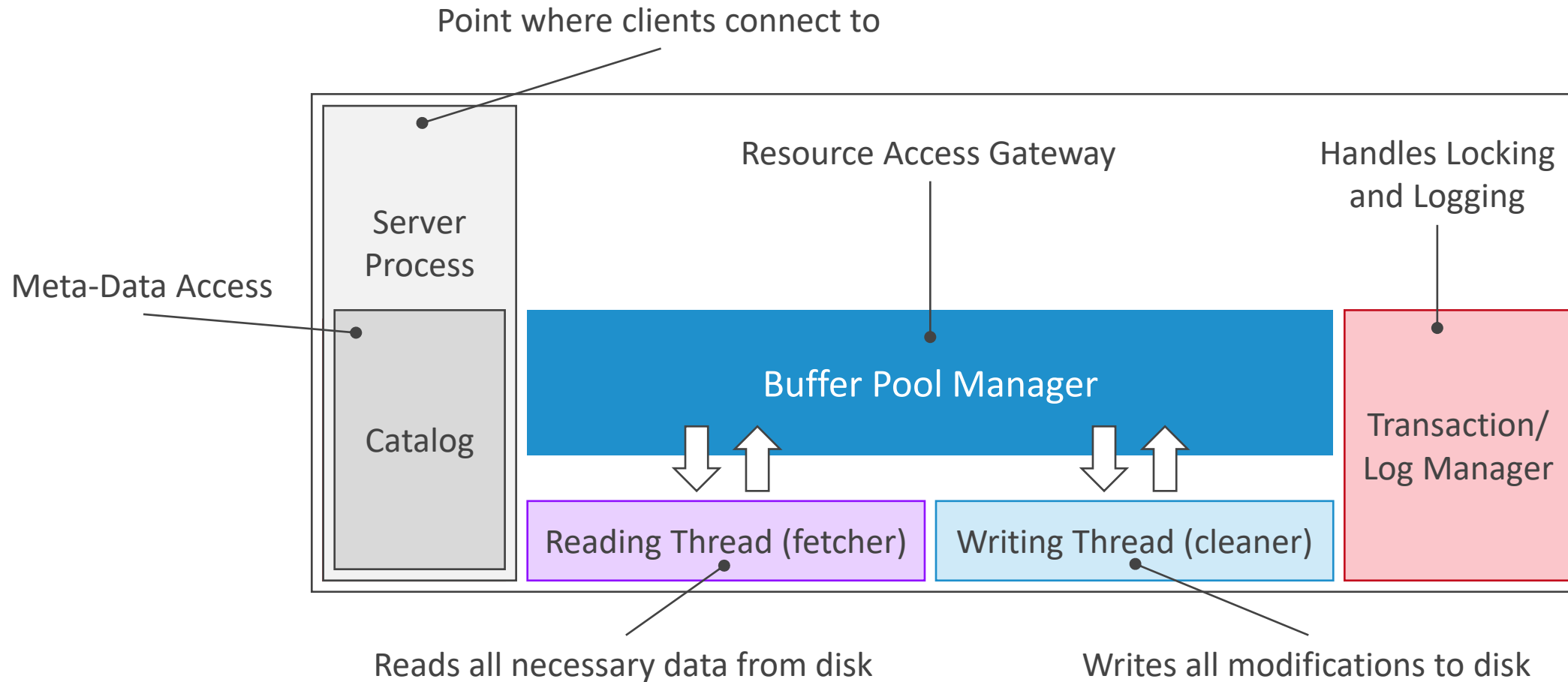
Recap: Retrieval of pages from disk

- Case 2: Page does not exist in the cache (cache miss).

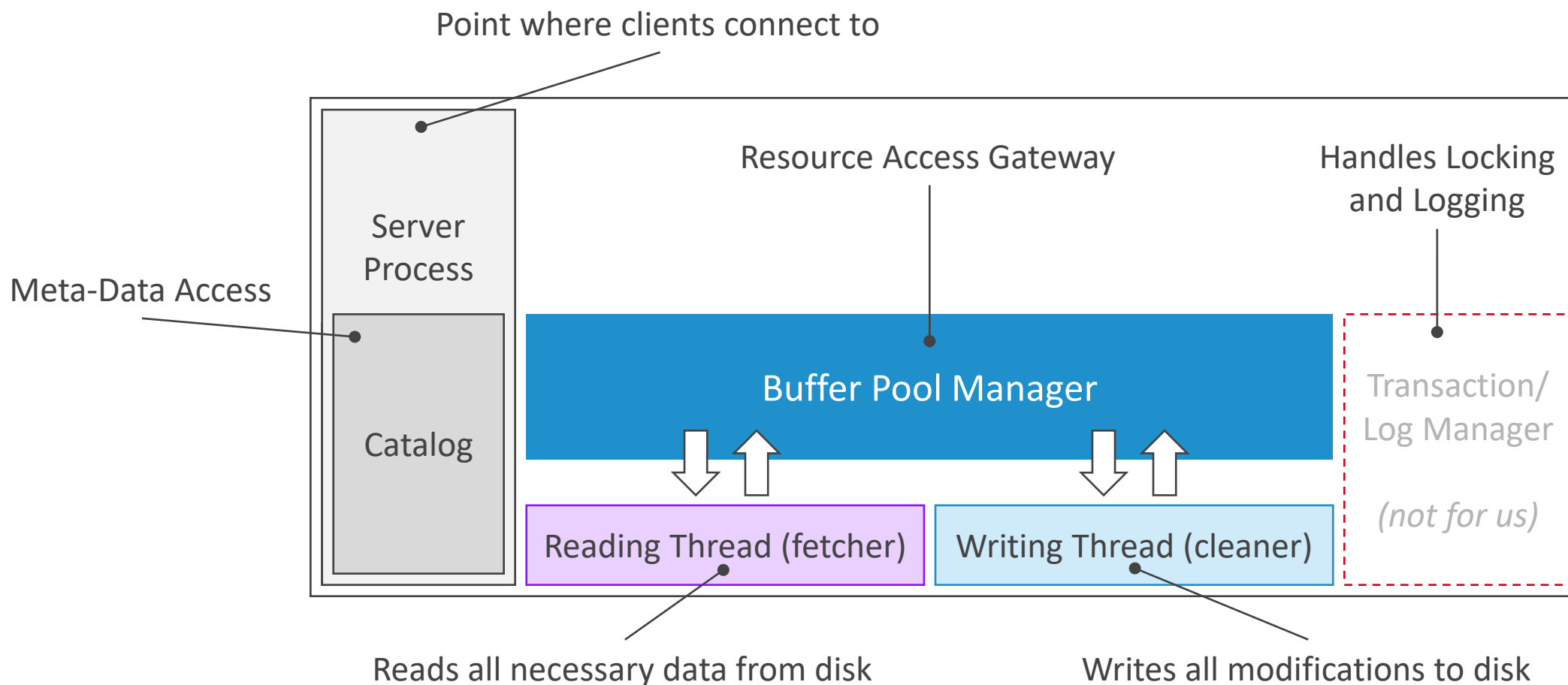


Process Model in a DBMS

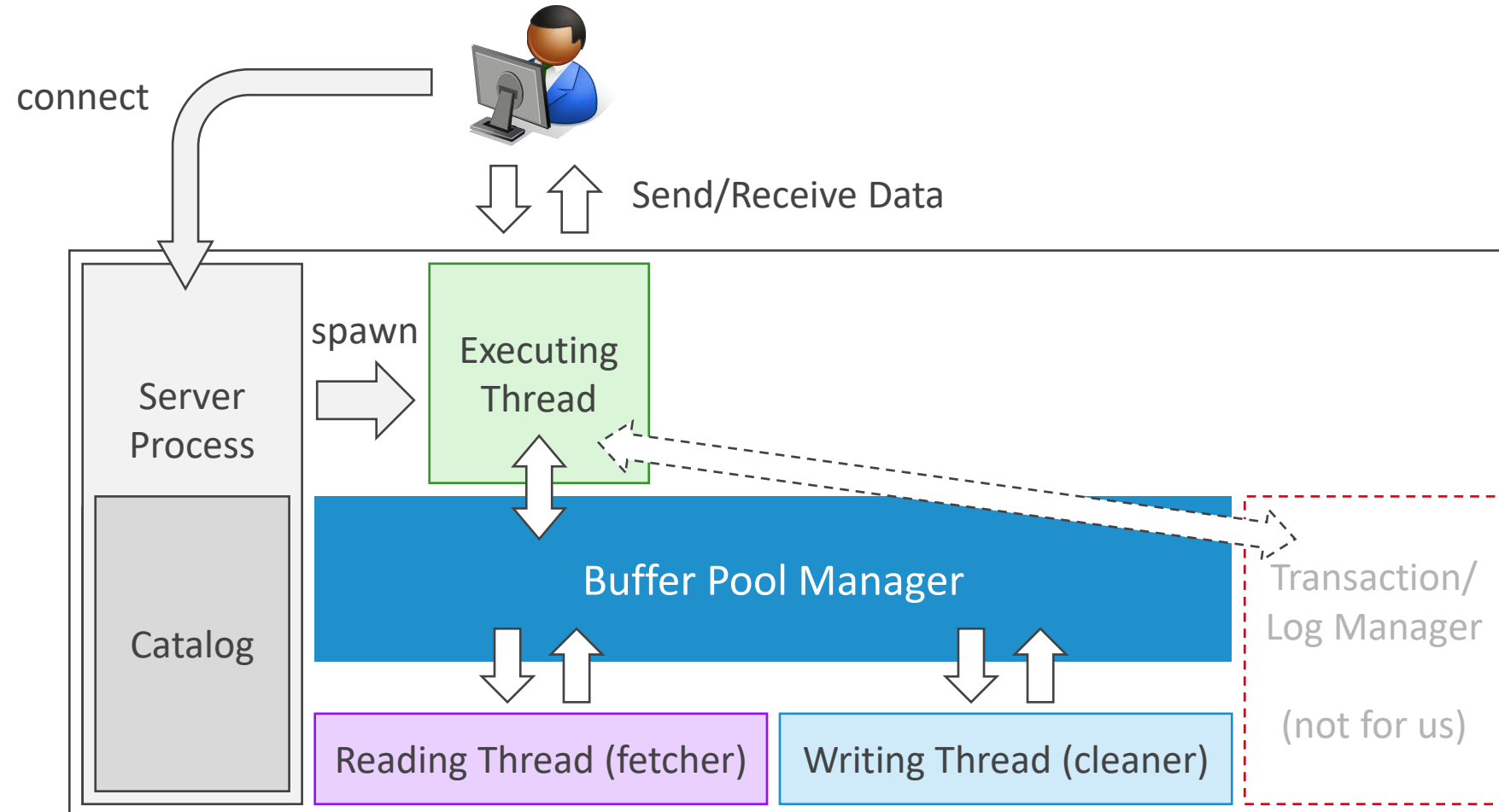
Process Model – System after Startup



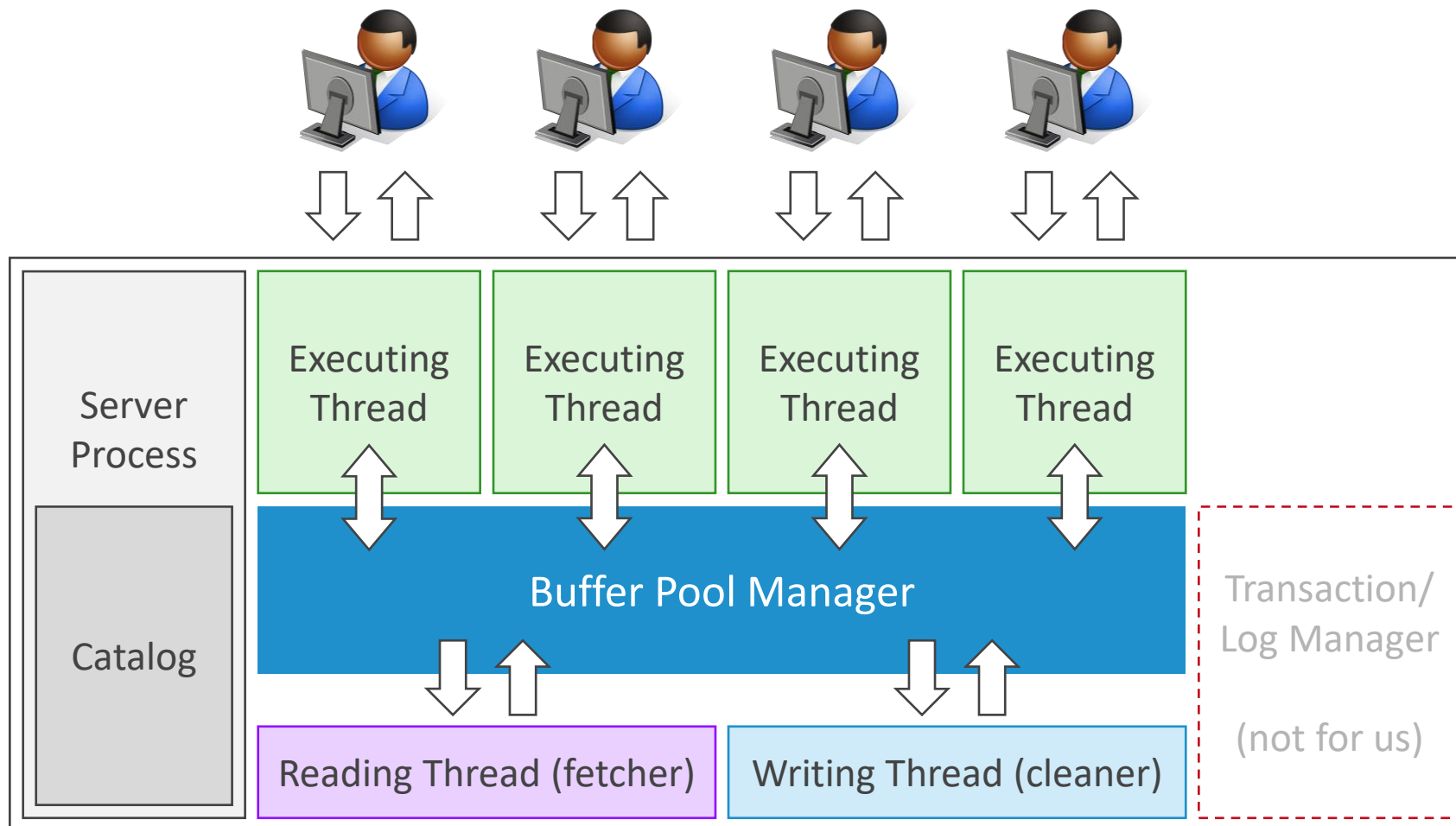
Process Model – System after Startup



Process Model – Connecting Client



Process Model – Multiple Queries



What Processes exist?

- This is the minimal number of processes (or threads) typically:
 - Instance owner (Main Process)
 - Connection receiver (where clients connect)
 - Query Execution Thread (do the processing, multiple for multi-core execution)
 - I/O Agents – reading and writing (may be combined into a single thread)
 - Transaction Manager (may consist of multiple threads for different subtasks)
- **WARNING: Highly simplified!!!**

The Buffer Pool Manager

- Central component through which all pages are requested.
- One instance of that component serves all requests from all concurrent queries.
 - Necessary to exploit the opportunity to share cached data.
- Owns the cache (or the caches).
- Usually employs sub-processes (or threads) to perform read and write I/O operations.
- Needs to perform efficient synchronization between the requests!

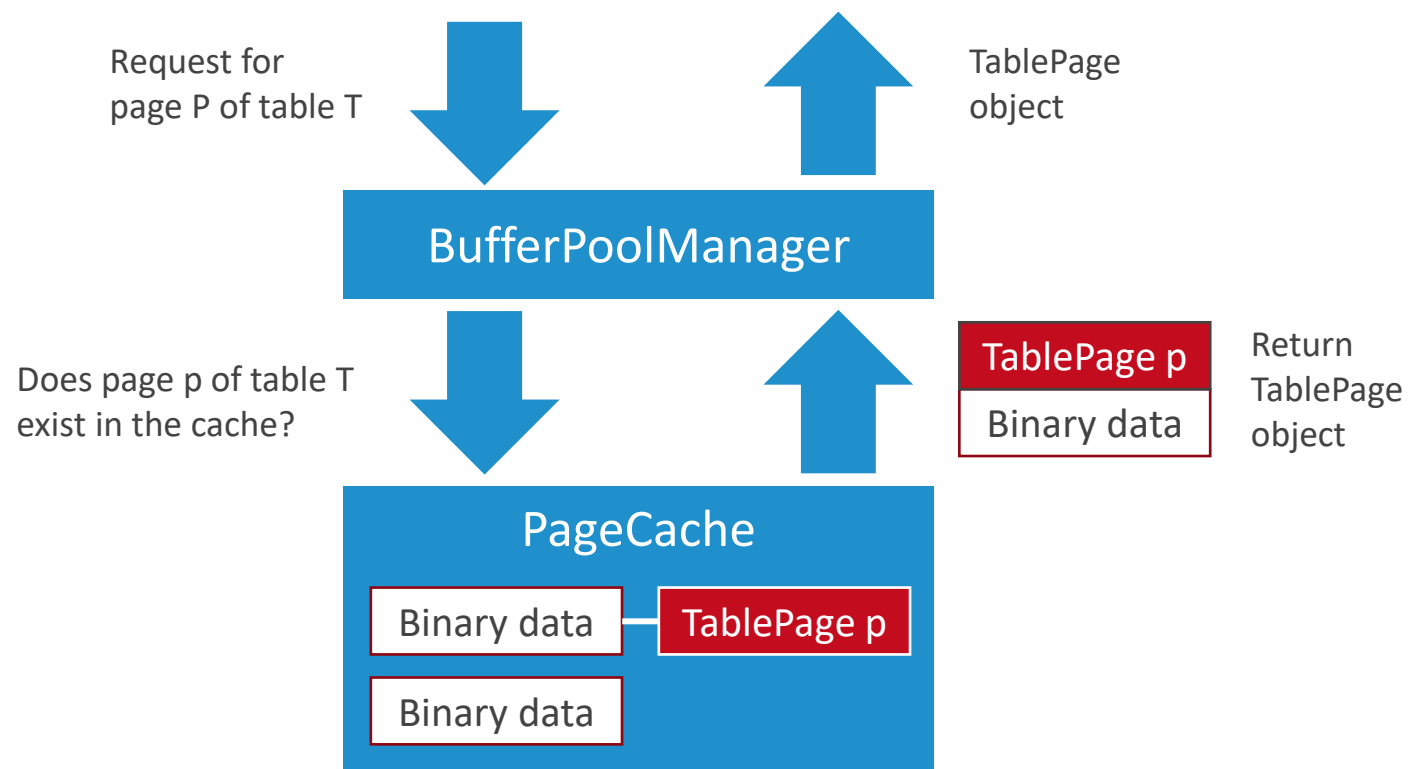
Resource Managers

- Resources (e.g. different tables and indexes) are registered at the Buffer Pool Manager.
- Every resource has a ResourceManager, which is a wrapper around the file for that resource.
- Relatively simple classes with methods for:
 - Obtaining the schema of the resource (Table- or IndexSchema)
 - Reading pages
 - Writing pages
 - Creating new pages (reserving and initializing new space in the file)
 - Getting the number of pages
- The methods that read and create the pages return Cacheable data objects.
 - For example, TableResourceManagers return TablePages.
- Fully transparent for the Buffer Pool Manager, with no case distinctions.

Important Operations

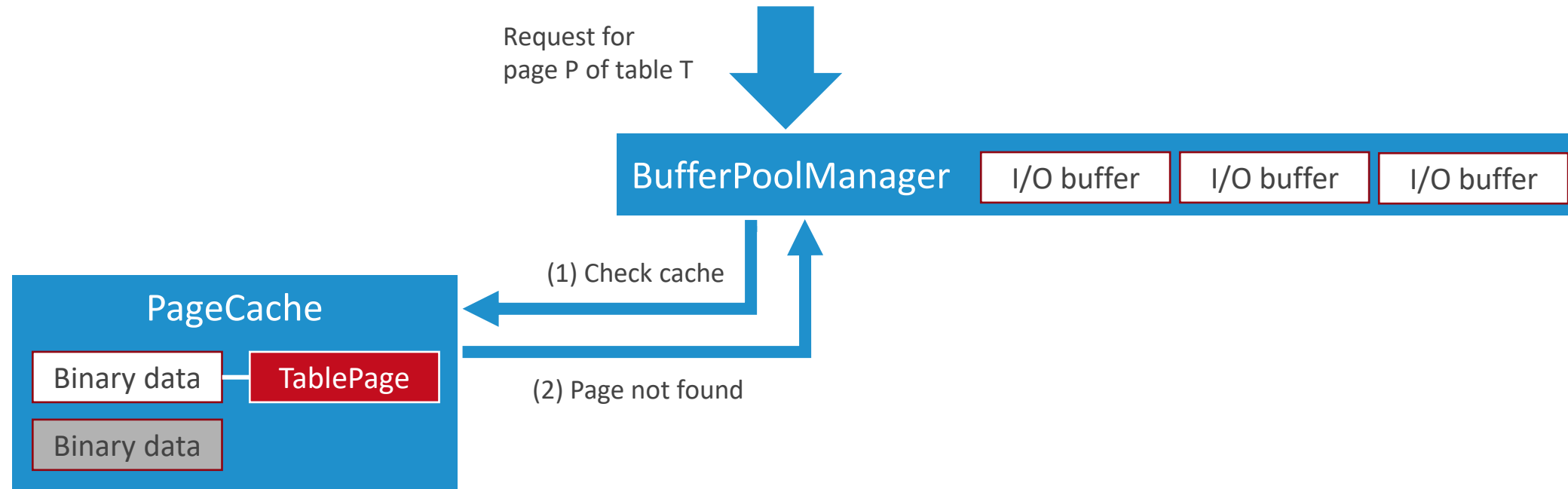
Retrieval of pages from the cache

- Case 1: Page exists in the cache (cache hit).



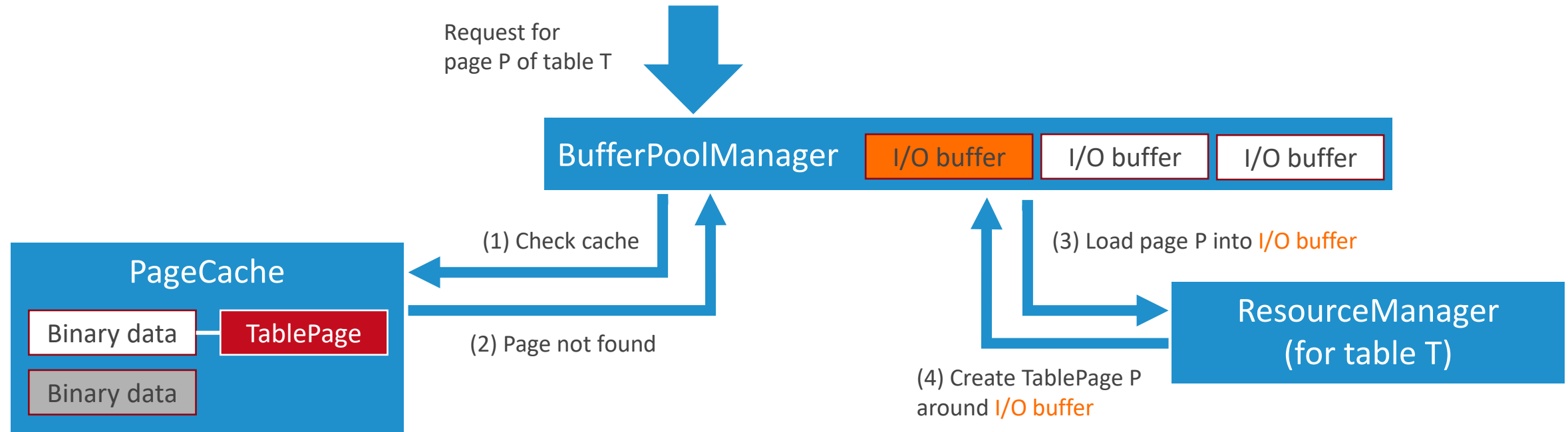
Recap: Retrieval of pages from disk

- Case 2: Page does not exist in the cache (cache miss).



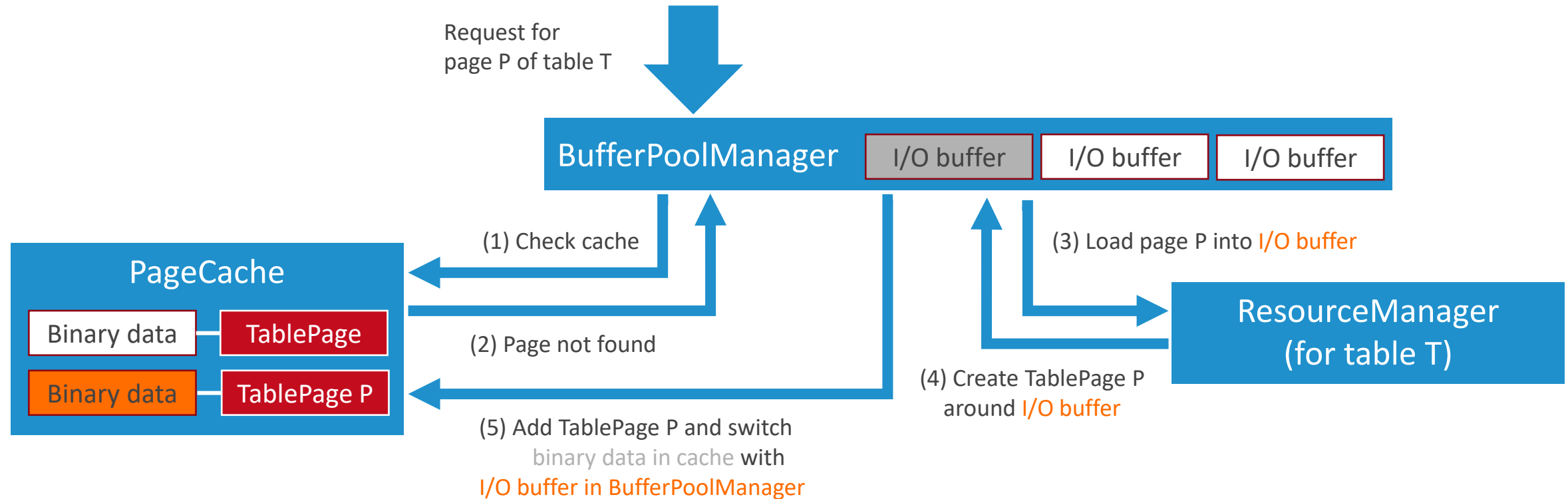
Recap: Retrieval of pages from disk

- Case 2: Page does not exist in the cache (cache miss).



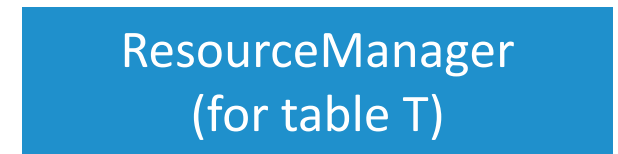
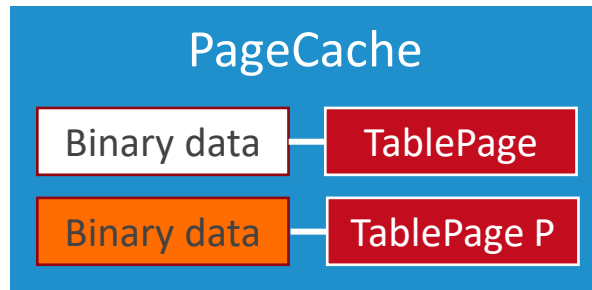
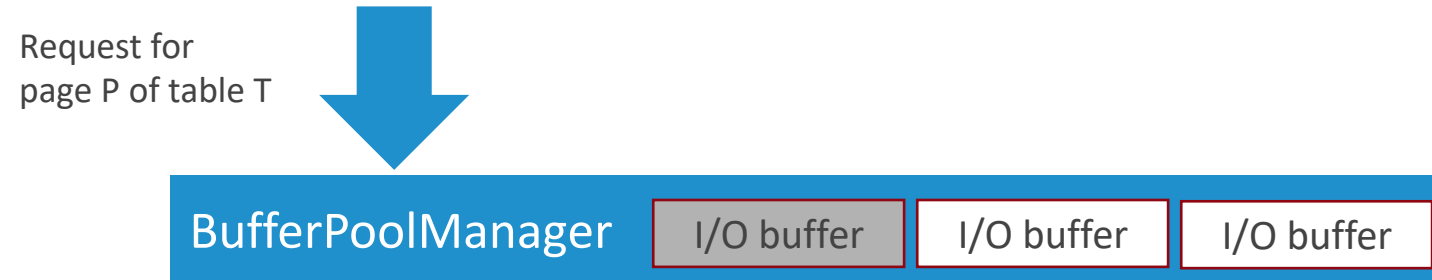
Recap: Retrieval of pages from disk

- Case 2: Page does not exist in the cache (cache miss).



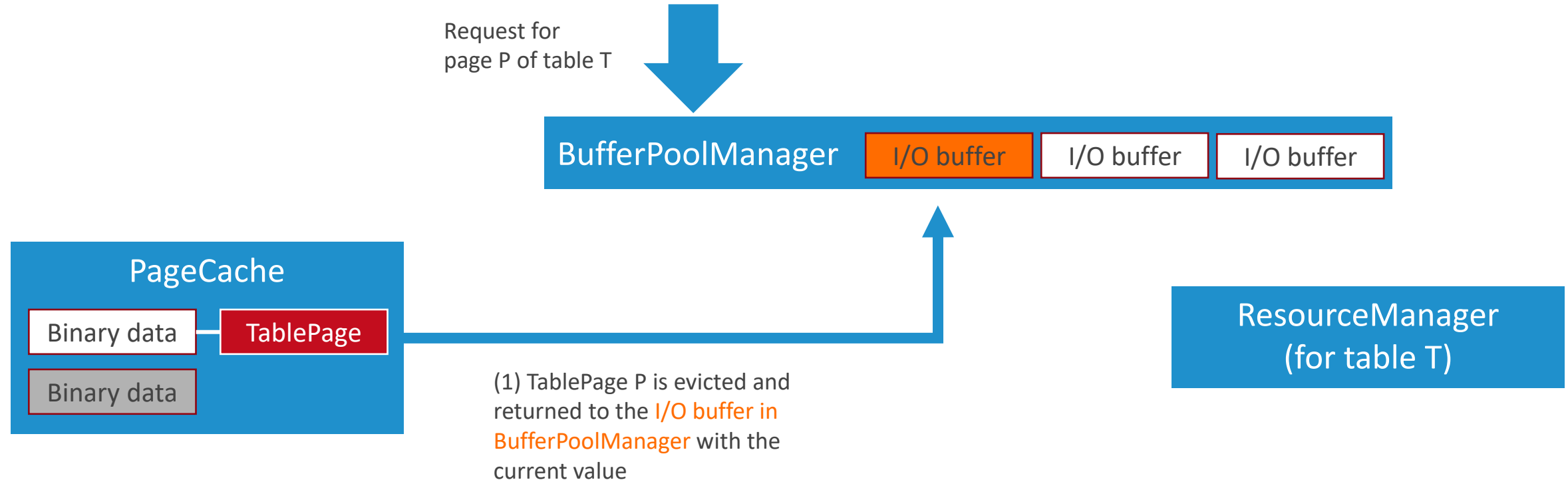
Retrieval of pages from disk

- Case 2: Page does not exist in the cache (cache miss).



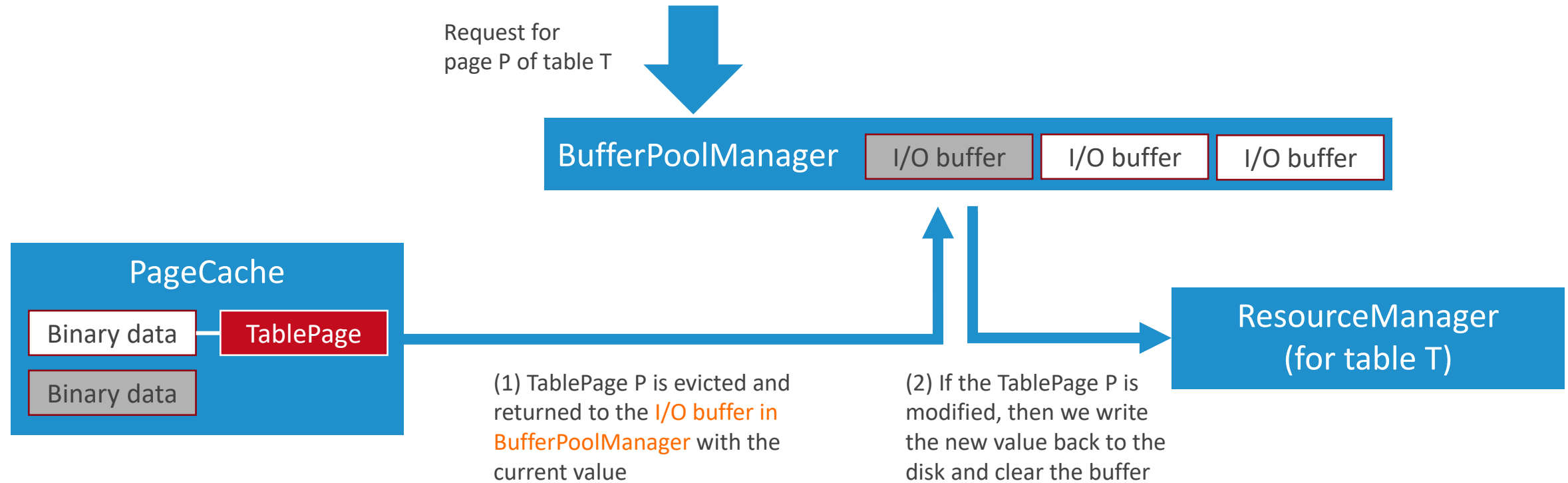
Writing modified pages back to disk

- Case 3: Page was modified and evicted from cache.



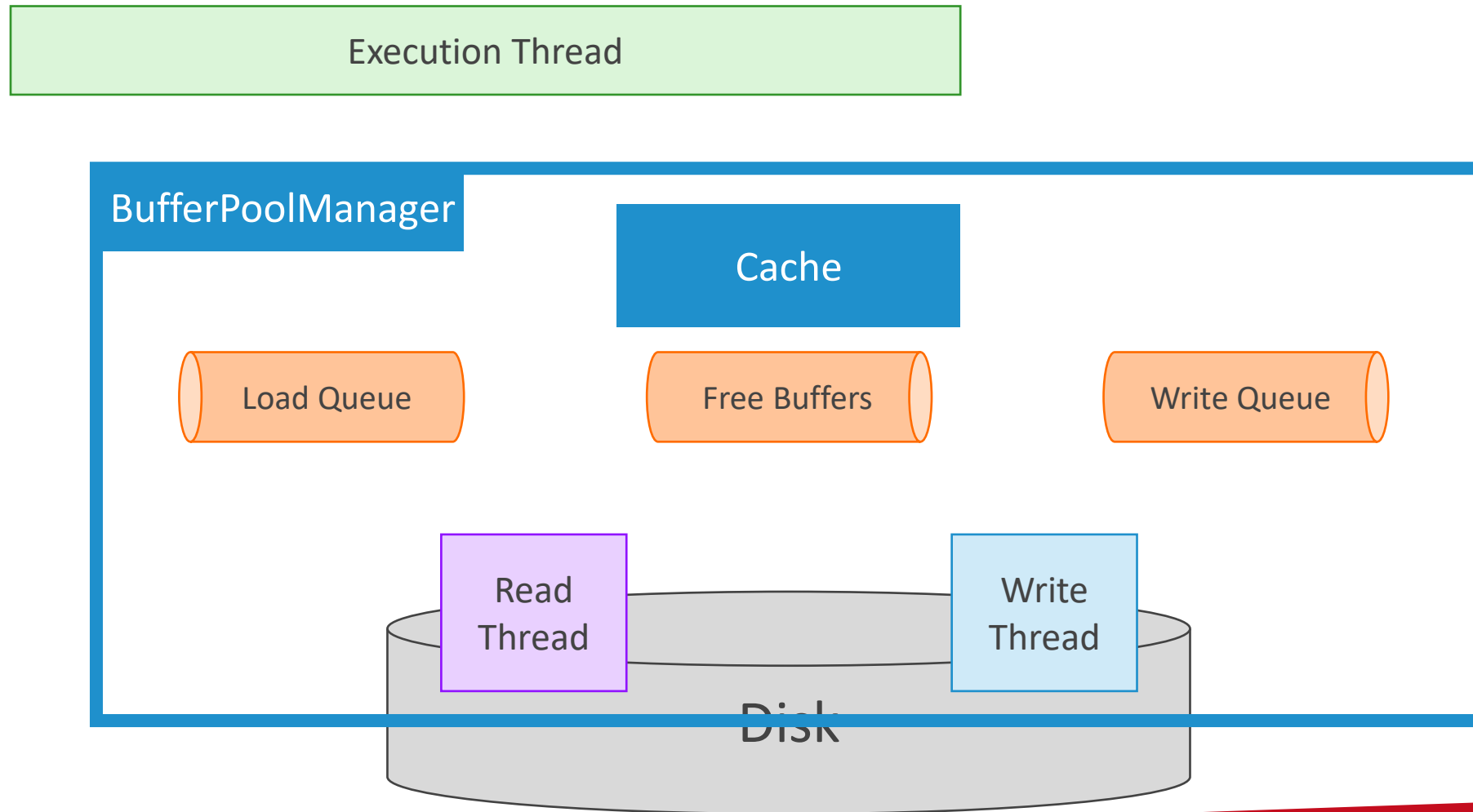
Writing modified pages back to disk

- Case 3: Page was modified and evicted from cache.

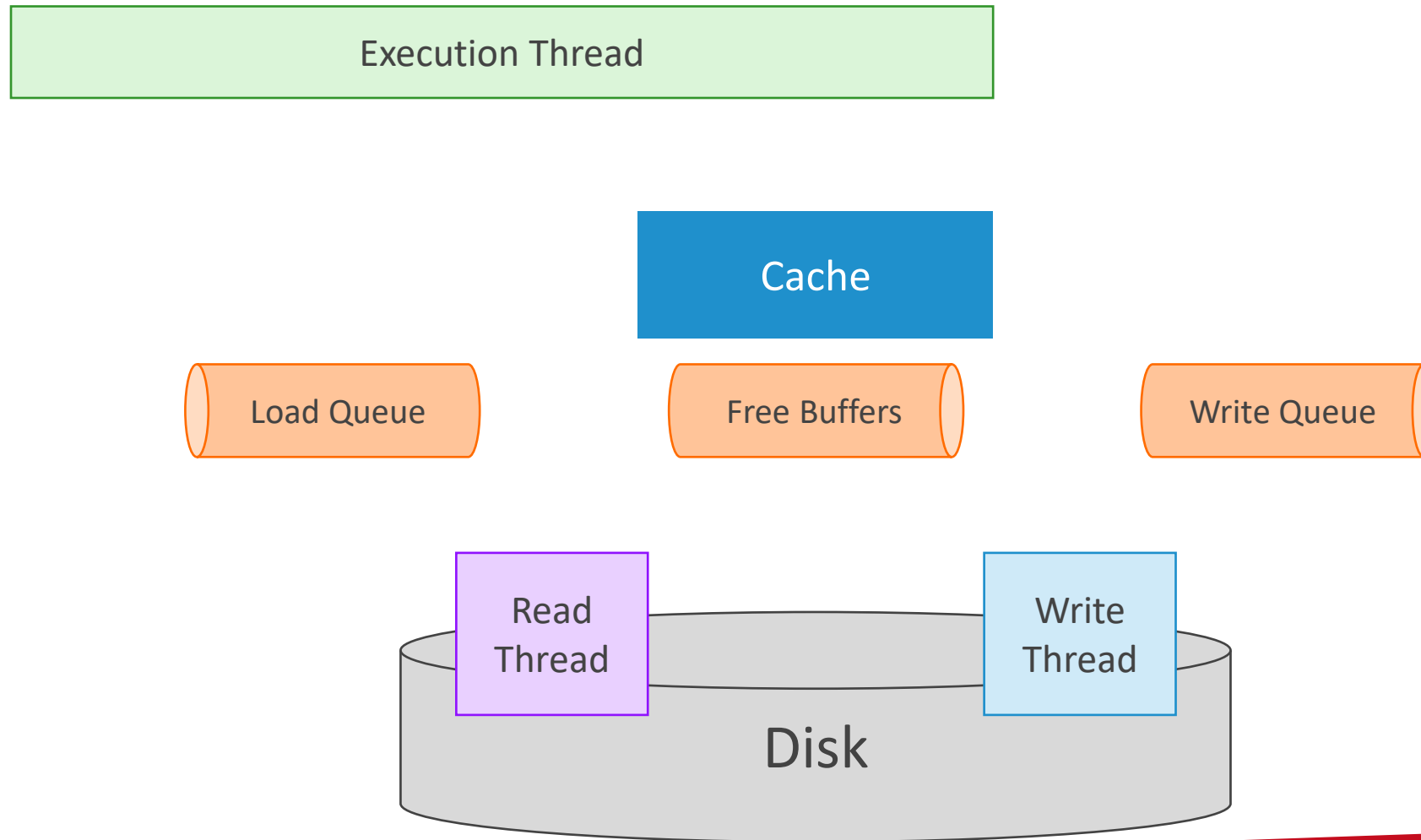


Implementation Design

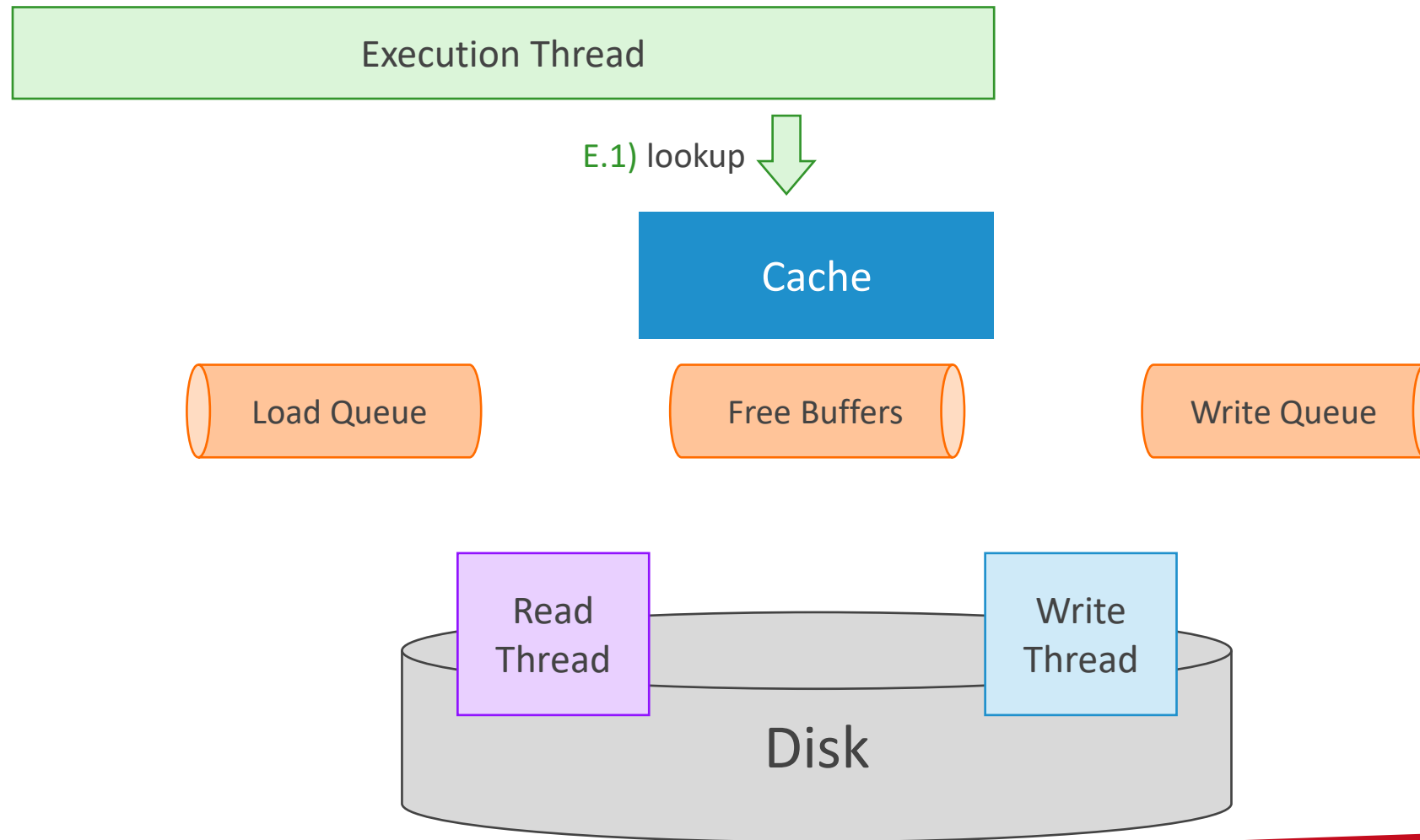
I/O in Detail – A queue model



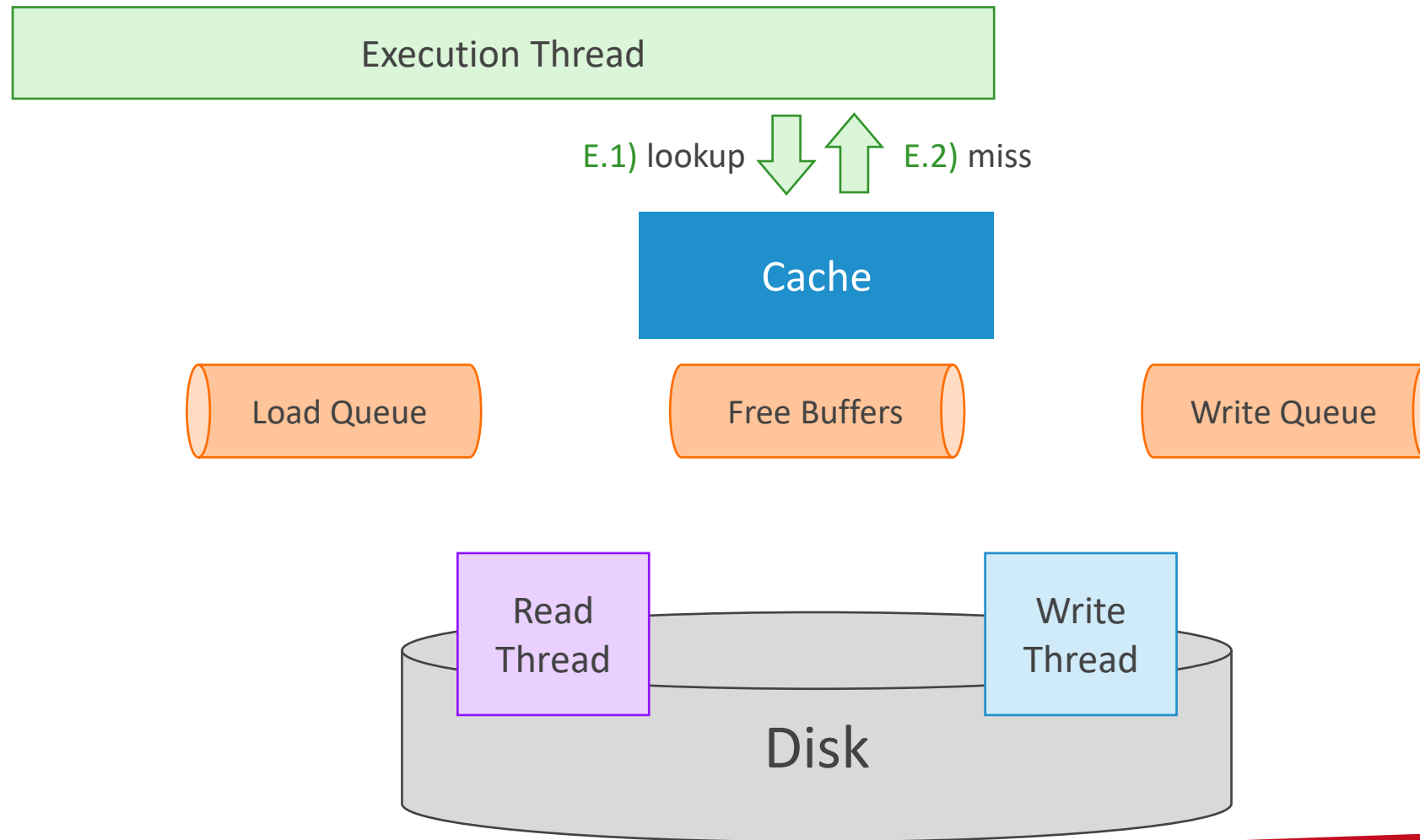
I/O in Detail – A queue model



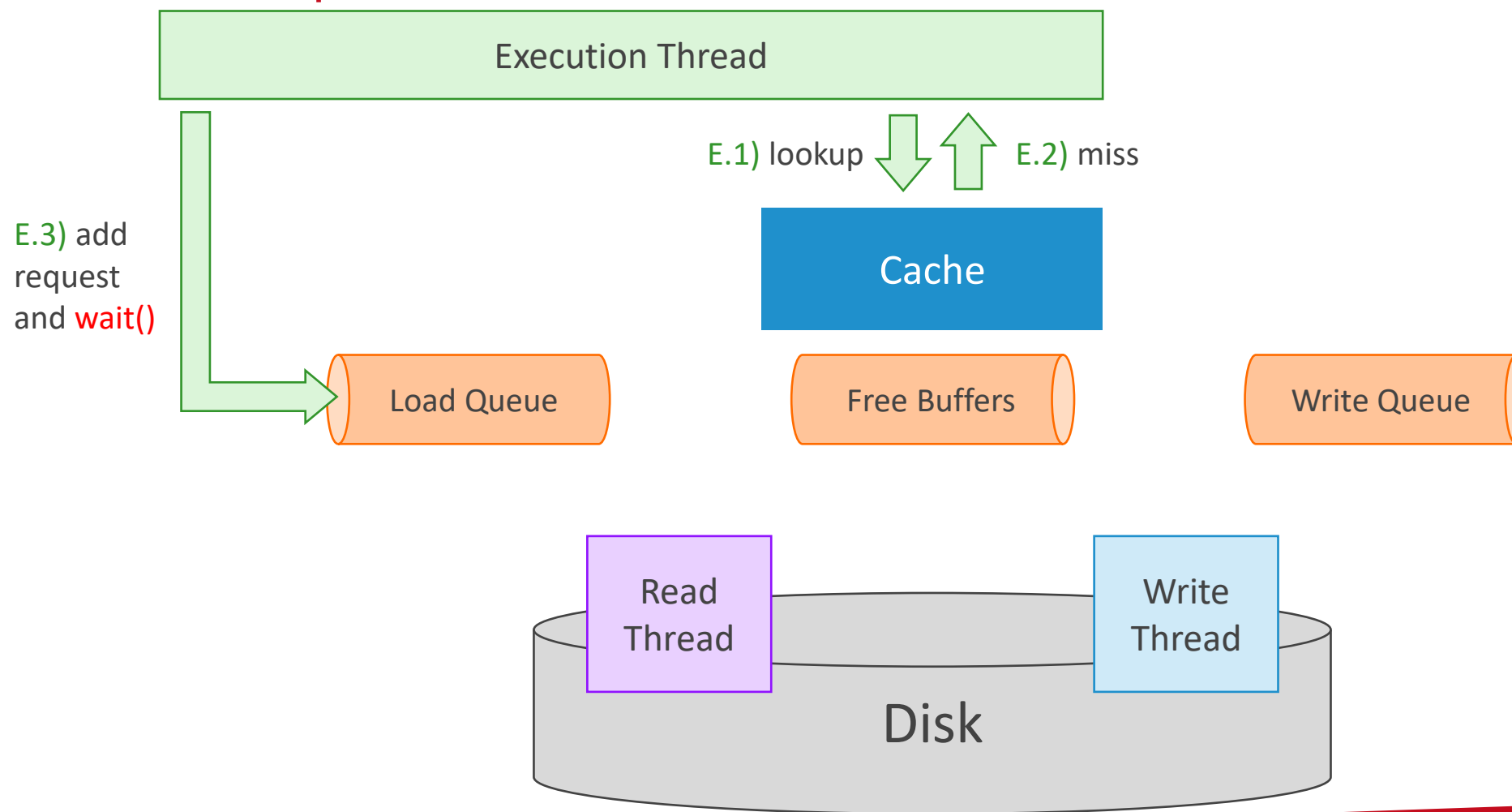
I/O in Detail – A queue model



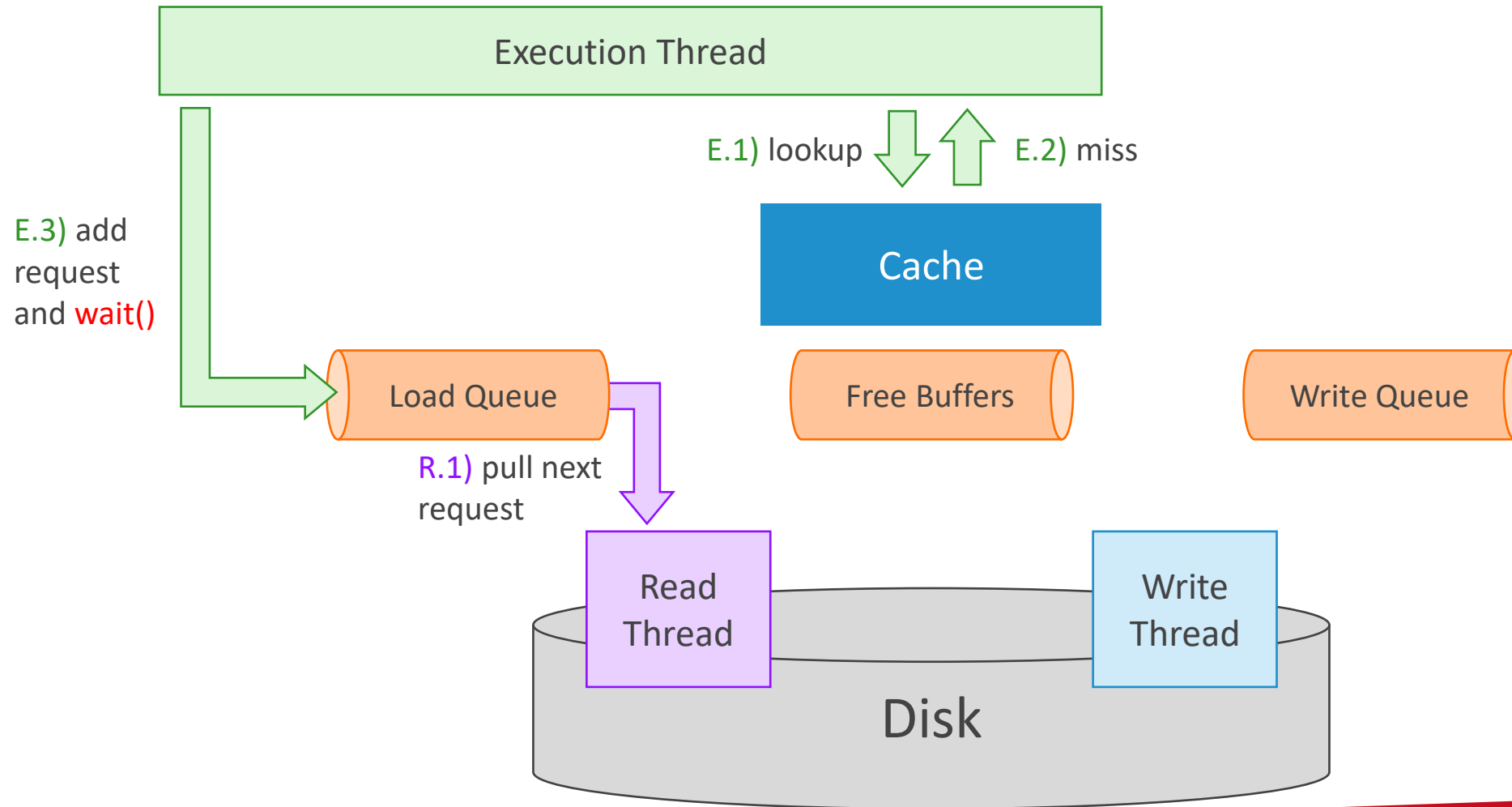
I/O in Detail – A queue model



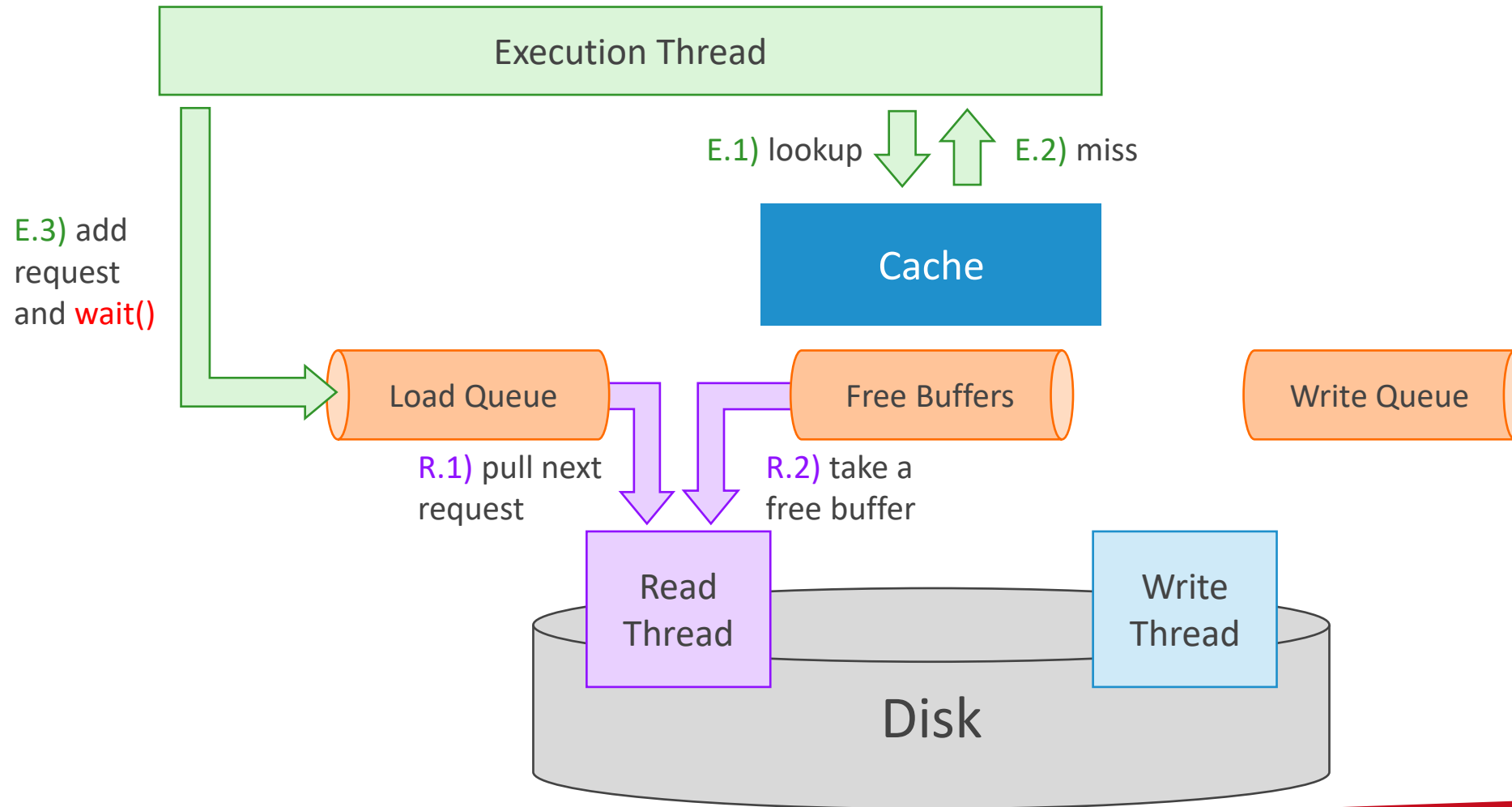
I/O in Detail – A queue model



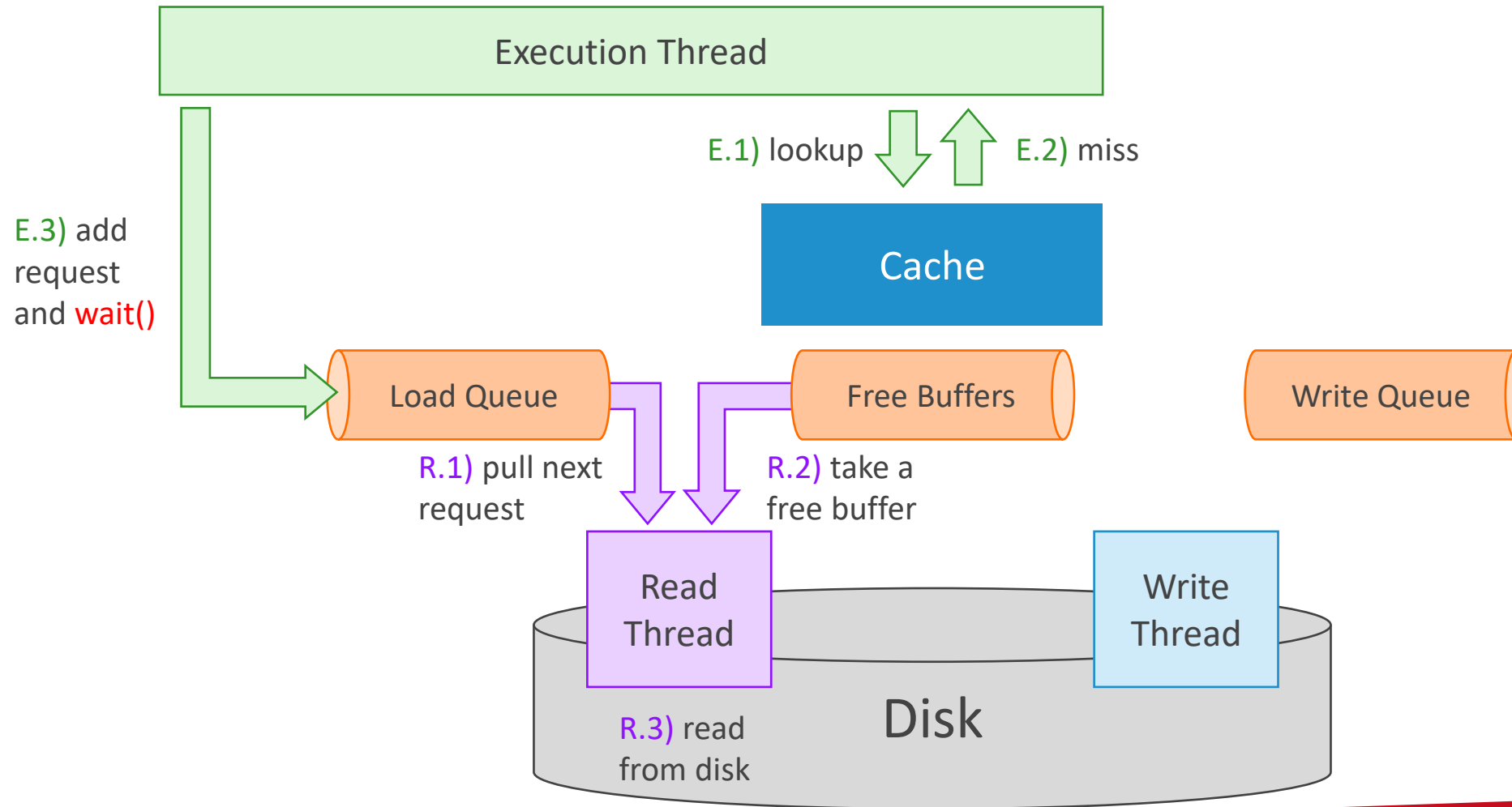
I/O in Detail – A queue model



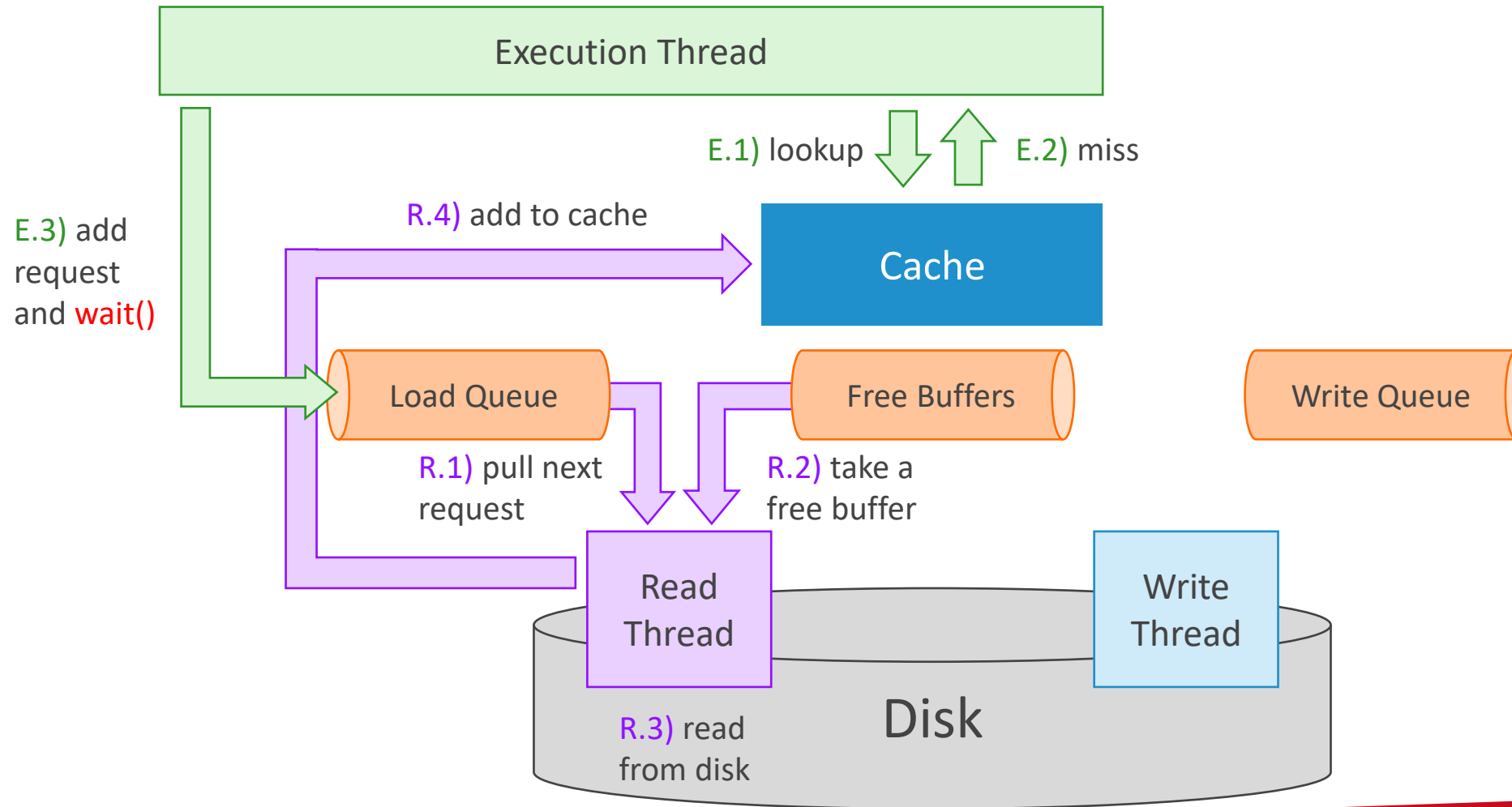
I/O in Detail – A queue model



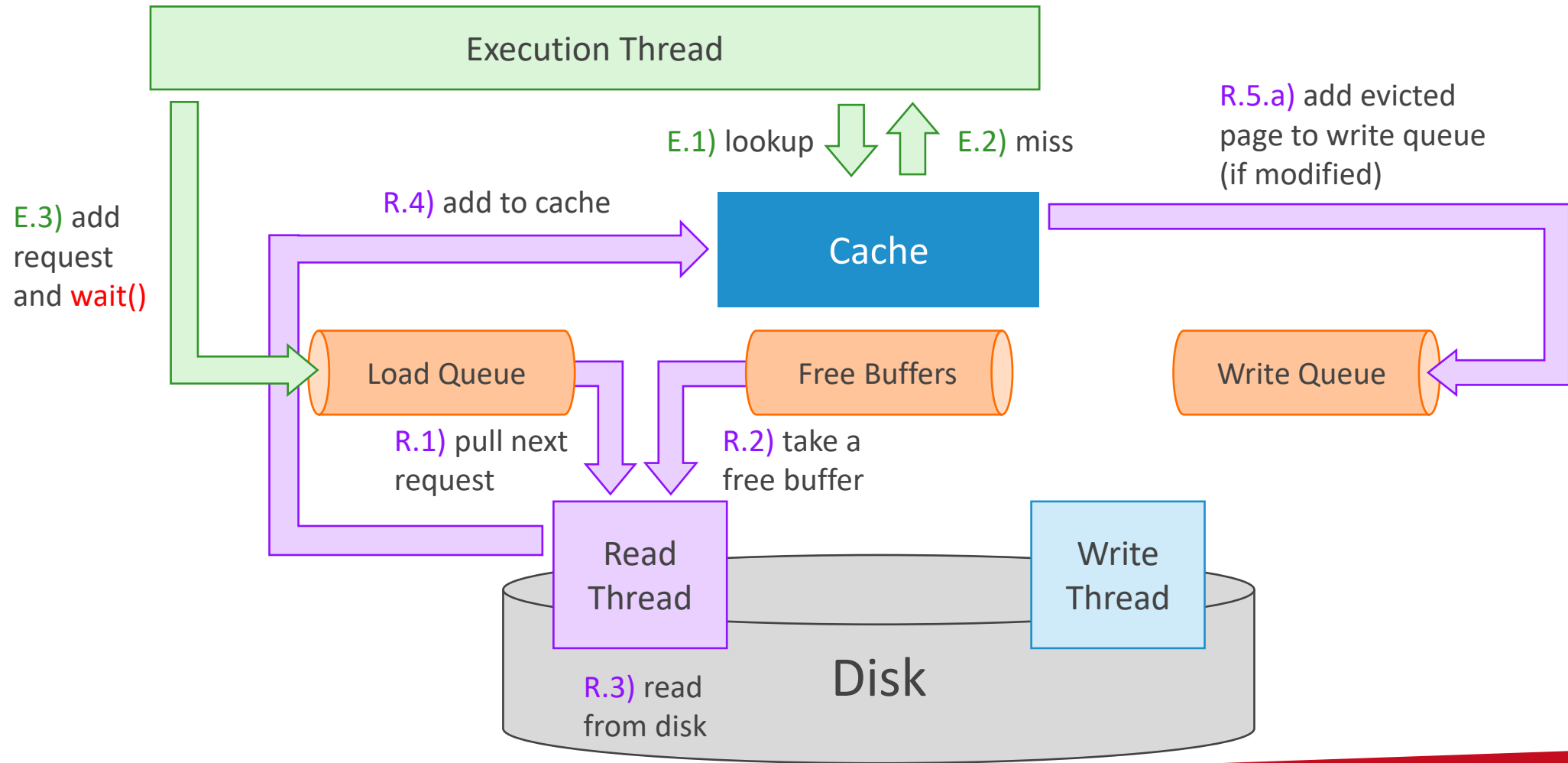
I/O in Detail – A queue model



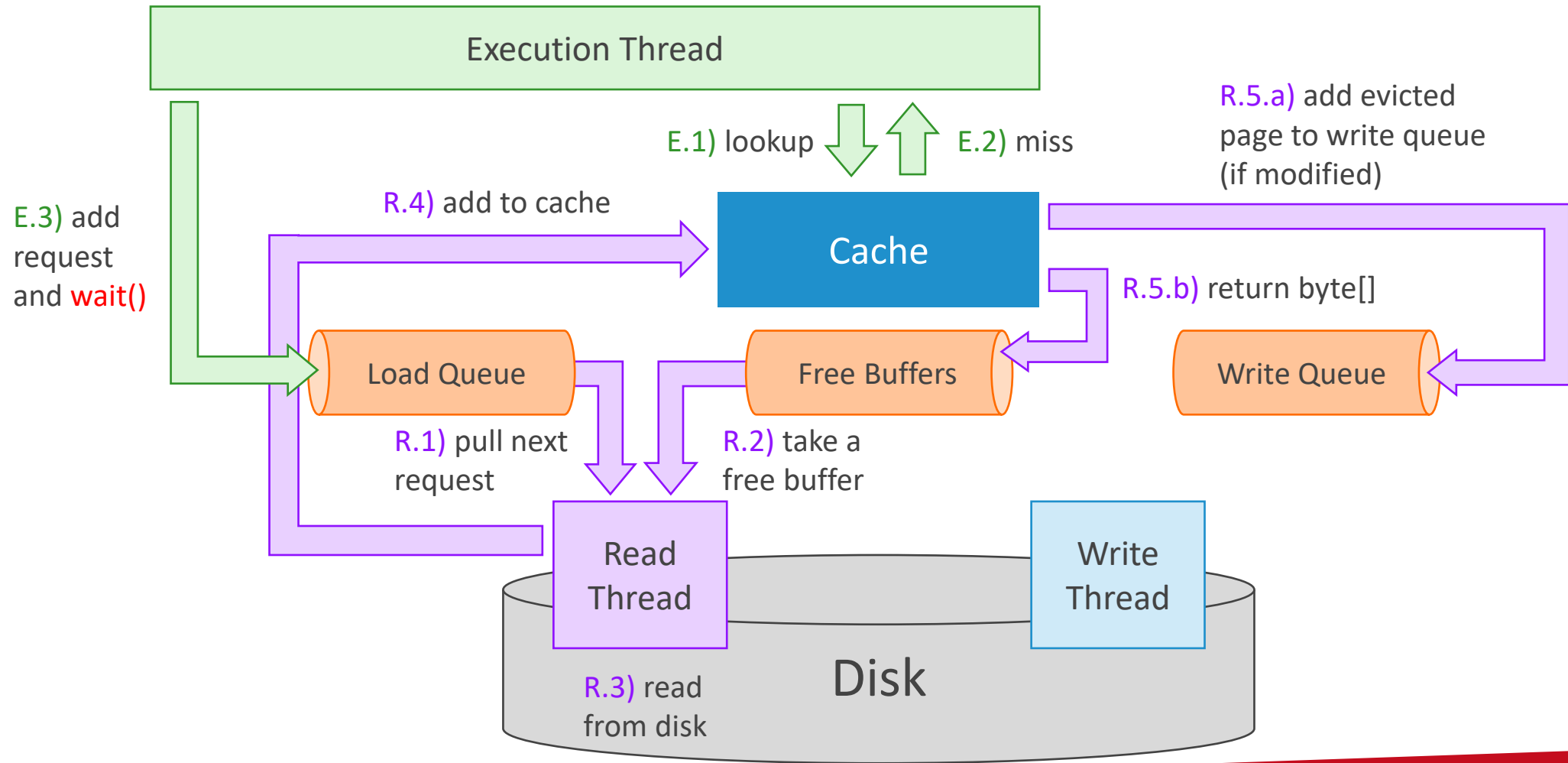
I/O in Detail – A queue model



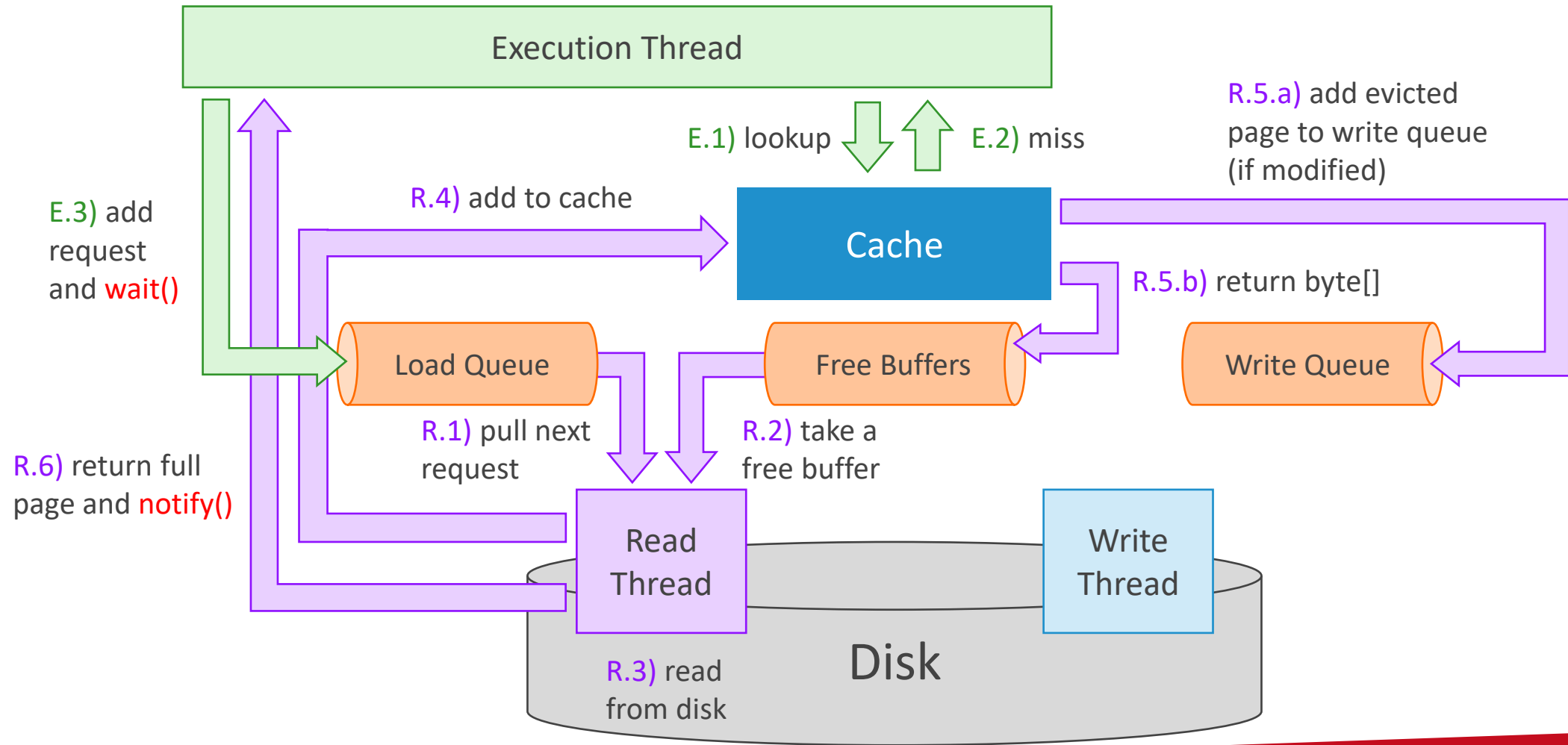
I/O in Detail – A queue model



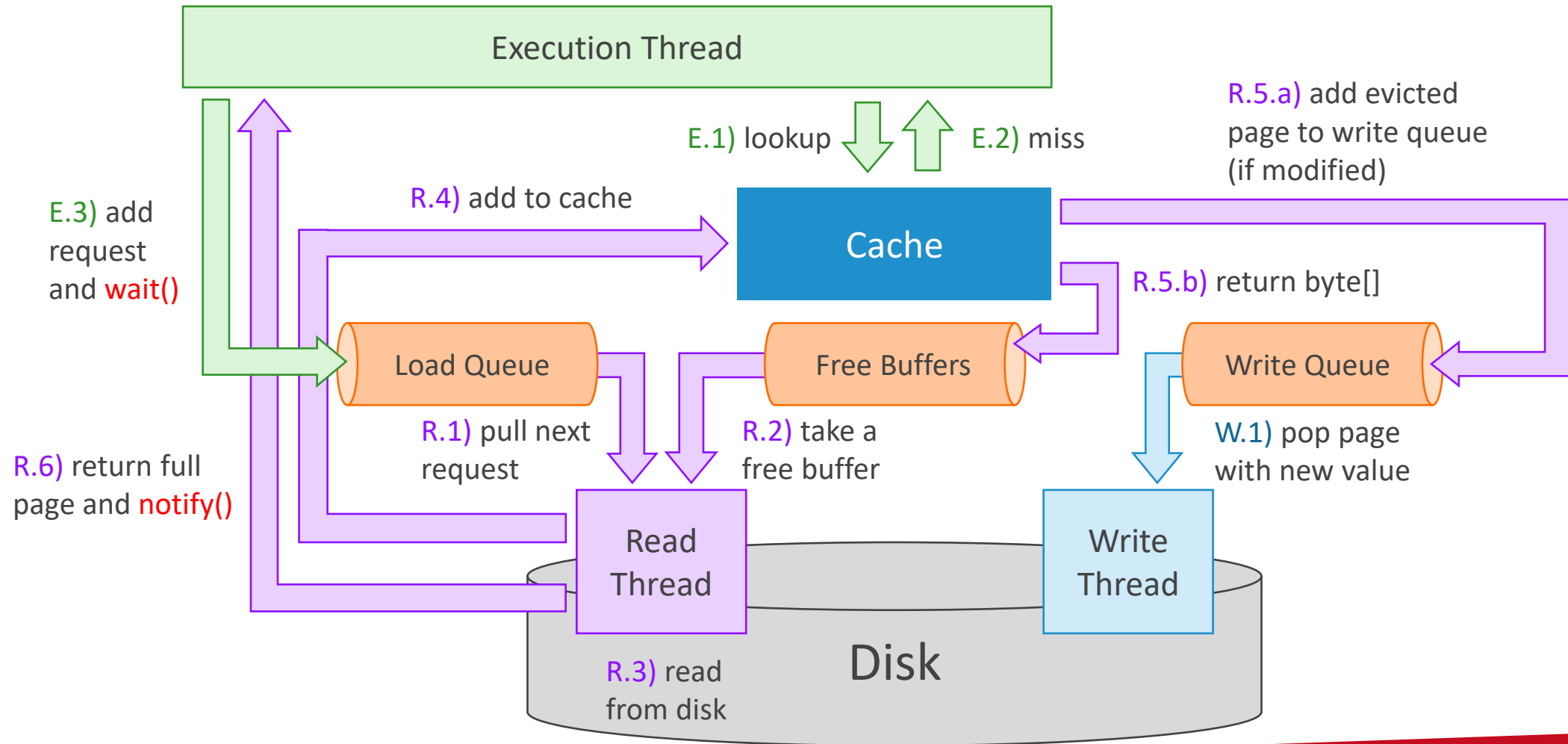
I/O in Detail – A queue model



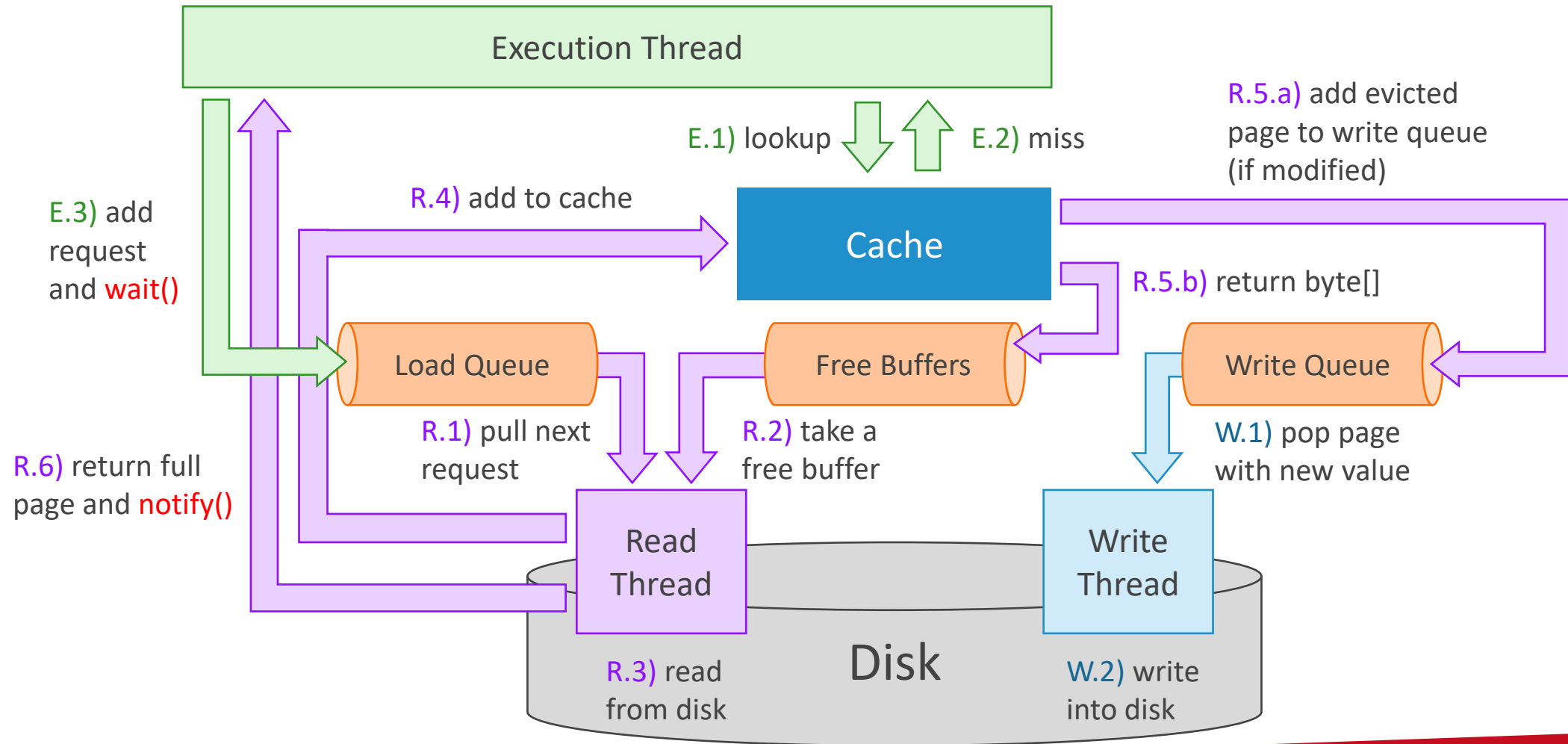
I/O in Detail – A queue model



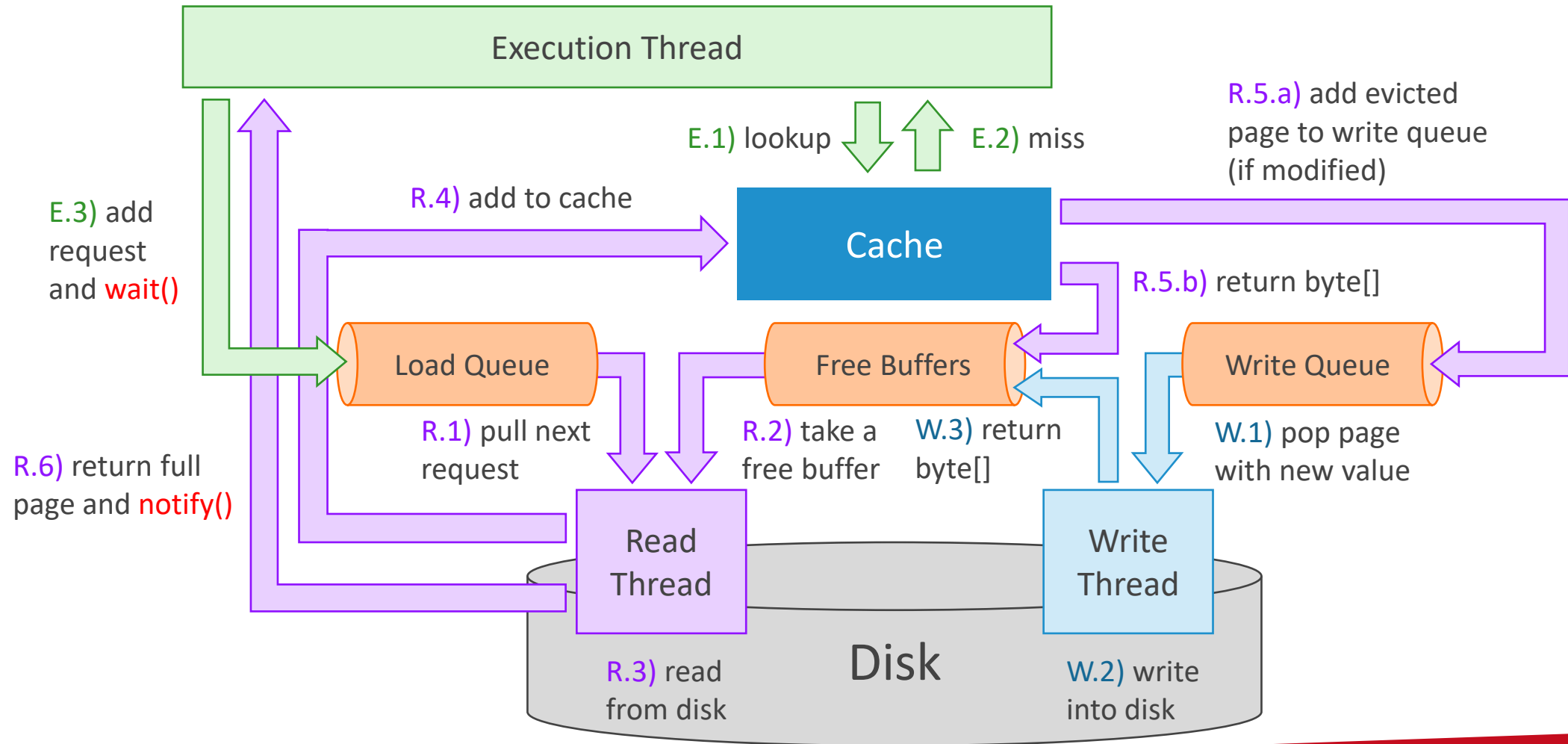
I/O in Detail – A queue model



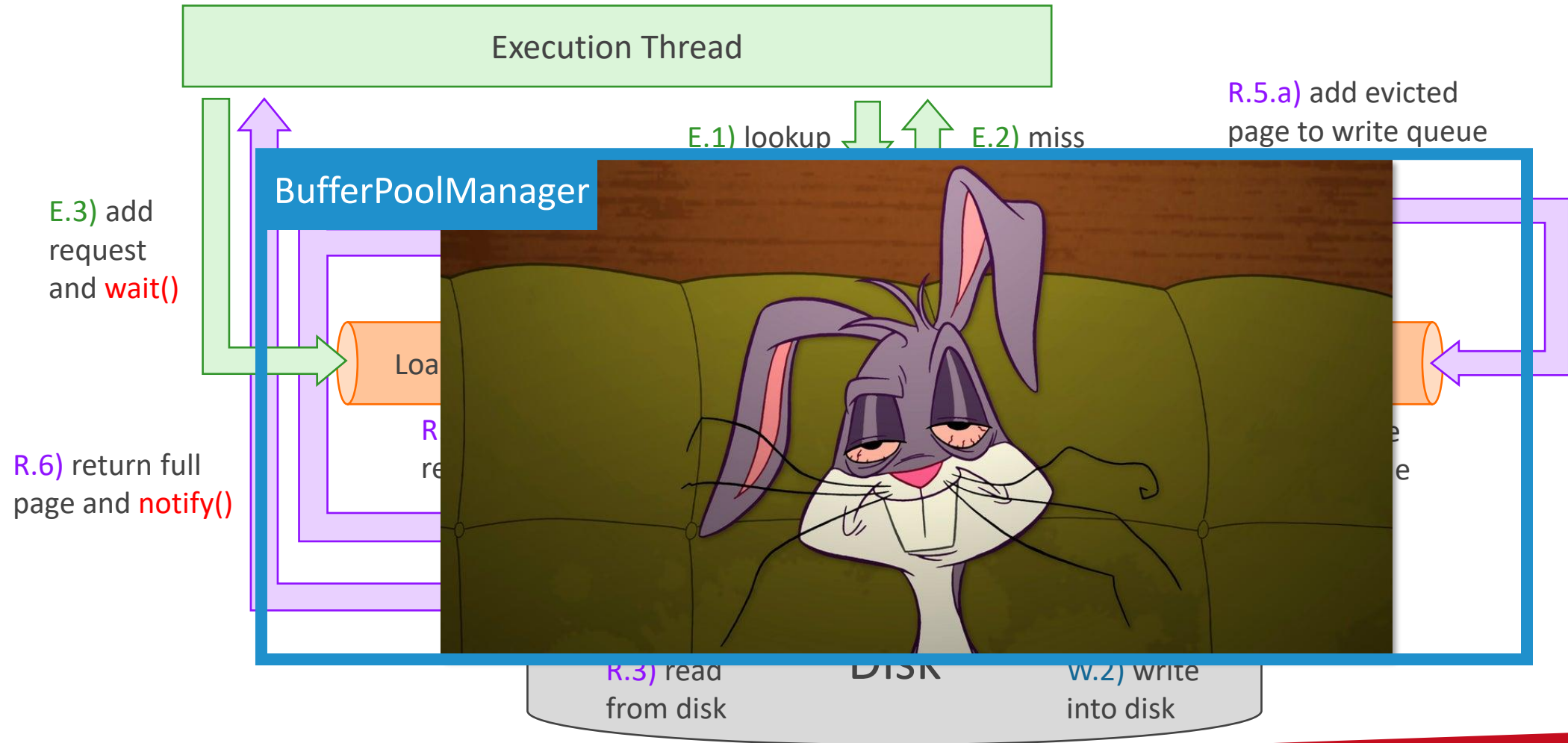
I/O in Detail – A queue model



I/O in Detail – A queue model

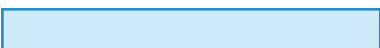


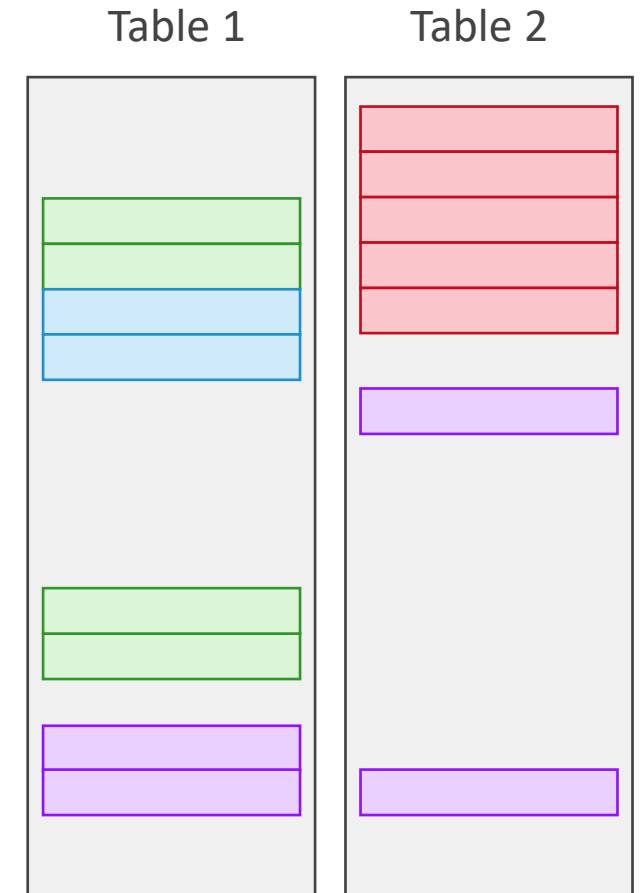
I/O in Detail – A queue model



Software Command Queuing

Making concurrent I/O requests more effective

- Assume the following case:
Reading from a table with multiple threads concurrently
 - Thread 1 
 - Thread 2 
 - Thread 3 
 - Thread 4 
- The threads issue requests to blocks in close distance
- How can it be done more efficiently?



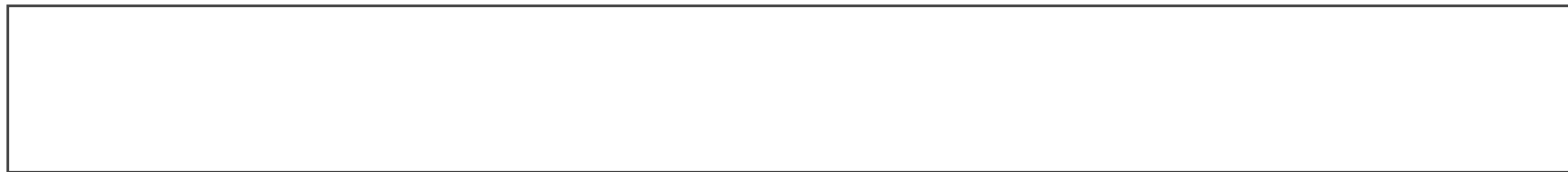
Software Command Queuing

- Assumption: All tables are clustered.
 - Requests to adjacent pages cause reads of physically-close blocks.
- Solution: Make the request queue **not** a simple FIFO queue but a queue that clusters close requests.
 - Requests are stored together if going to the same file.
 - Have page numbers with a difference of at most n (for example, $n = 10$).
- Close requests end up in the same section of the queue.
- Section is ordered after page number to make sure that.
- The **request queue becomes a queue-of-queues.**
- **Warning:** Make sure that the sub-queues are limited in size.
 - First sub-queue may sometimes grow for a long time (table scan), delaying other requests for too long.

Software Command Queuing Illustrated

- In this example, we assume that page requests that are at most 2 pages away from each other are grouped
- No sub-queue may be longer than 4 entries

```
List<QueueEntry> requestQueue;
```

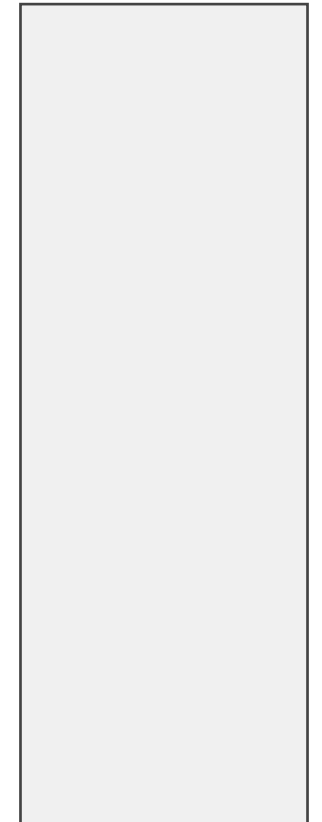


Load/Write Queue

Table 1



Table 2



Software Command Queuing Illustrated

- In this example, we assume that page requests that are at most 2 pages away from each other are grouped
- No sub-queue may be longer than 4 entries

```
List<QueueEntry> requestQueue;
```

```
List<List<QueueEntry>> requestQueue;
```



Load/Write Queue

Table 1



Table 2



Software Command Queuing Illustrated

- In this example, we assume that page requests that are at most 2 pages away from each other are grouped
- No sub-queue may be longer than 4 entries

```
List<QueueEntry> requestQueue;
```

```
List<List<QueueEntry>> requestQueue;
```



Load/Write Queue

Table 1

3

Table 2

1
2

Software Command Queuing Illustrated

- In this example, we assume that page requests that are at most 2 pages away from each other are grouped
- No sub-queue may be longer than 4 entries

```
List<QueueEntry> requestQueue;
```

```
List<List<QueueEntry>> requestQueue;
```



Load/Write Queue

Table 1

3

Table 2

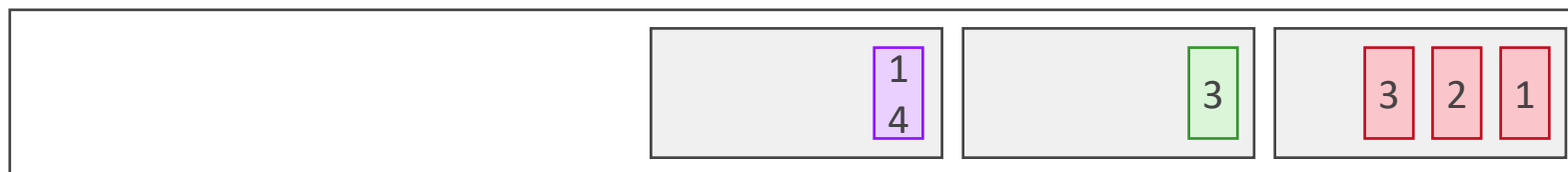
1
2
3

Software Command Queuing Illustrated

- In this example, we assume that page requests that are at most 2 pages away from each other are grouped
- No sub-queue may be longer than 4 entries

```
List<QueueEntry> requestQueue;
```

```
List<List<QueueEntry>> requestQueue;
```



Load/Write Queue

Table 1

3

Table 2

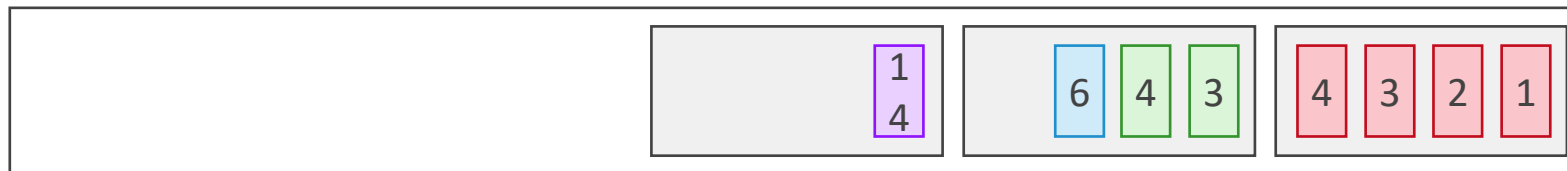
1
2
3
14

Software Command Queuing Illustrated

- In this example, we assume that page requests that are at most 2 pages away from each other are grouped
- No sub-queue may be longer than 4 entries

```
List<QueueEntry> requestQueue;
```

```
List<List<QueueEntry>> requestQueue;
```



Load/Write Queue

Table 1

3
4
6

Table 2

1
2
3
4
14

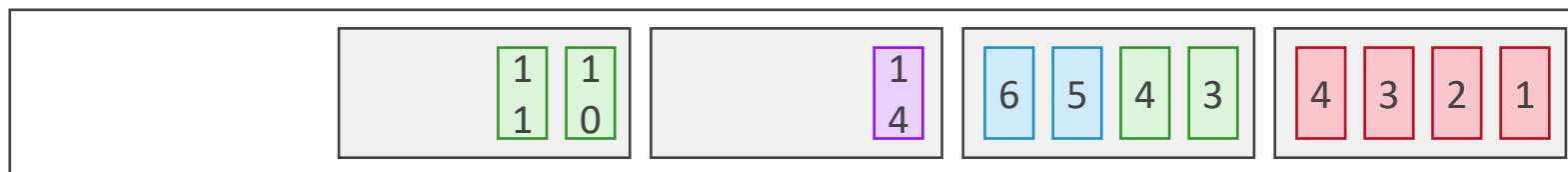
Load/Write Queue

Software Command Queuing Illustrated

- In this example, we assume that page requests that are at most 2 pages away from each other are grouped
- No sub-queue may be longer than 4 entries

```
List<QueueEntry> requestQueue;
```

```
List<List<QueueEntry>> requestQueue;
```



Load/Write Queue

Table 1

3
4
5
6
10
11

Table 2

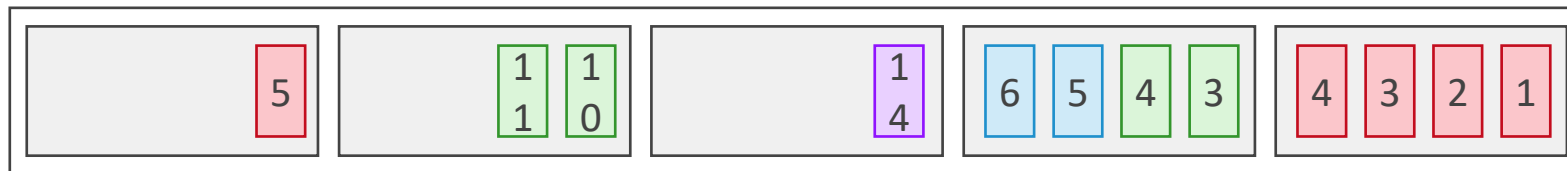
1
2
3
4
14

Software Command Queuing Illustrated

- In this example, we assume that page requests that are at most 2 pages away from each other are grouped
- No sub-queue may be longer than 4 entries

```
List<QueueEntry> requestQueue;
```

```
List<List<QueueEntry>> requestQueue;
```



Load/Write Queue

Table 1

3
4
5
6
10
11

Table 2

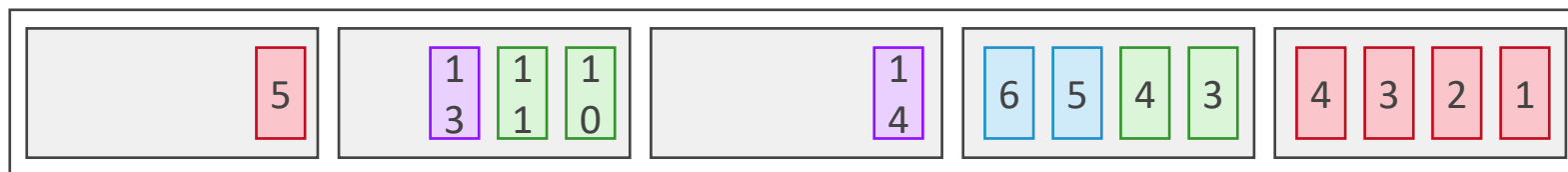
1
2
3
4
5
14

Software Command Queuing Illustrated

- In this example, we assume that page requests that are at most 2 pages away from each other are grouped
- No sub-queue may be longer than 4 entries

```
List<QueueEntry> requestQueue;
```

```
List<List<QueueEntry>> requestQueue;
```



Load/Write Queue

Table 1

3
4
5
6
10
11
13

Table 2

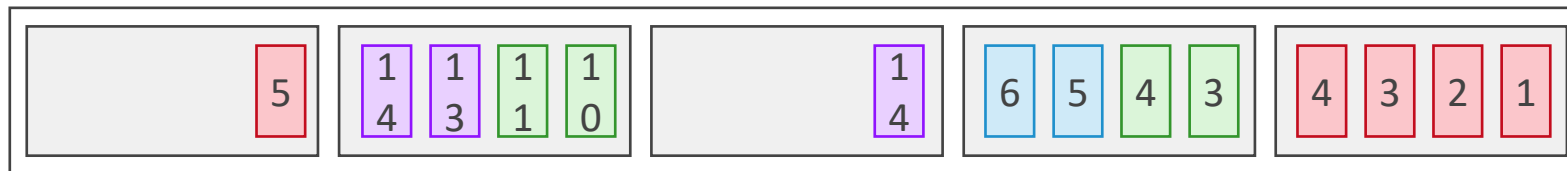
1
2
3
4
5
14

Software Command Queuing Illustrated

- In this example, we assume that page requests that are at most 2 pages away from each other are grouped
- No sub-queue may be longer than 4 entries

```
List<QueueEntry> requestQueue;
```

```
List<List<QueueEntry>> requestQueue;
```



Load/Write Queue

Table 1

3
4
5
6
10
11
13
14

Table 2

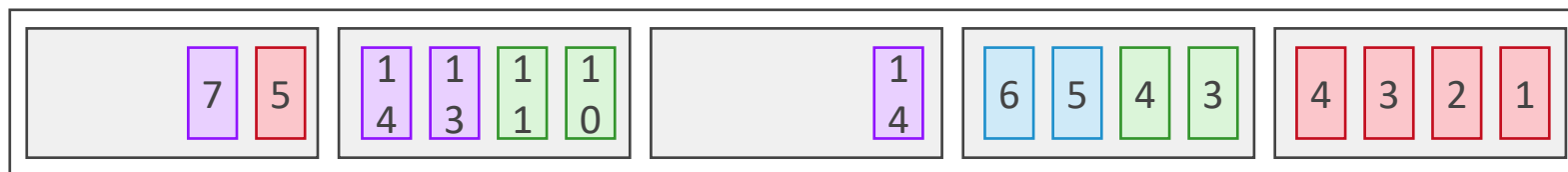
1
2
3
4
5
14

Software Command Queuing Illustrated

- In this example, we assume that page requests that are at most 2 pages away from each other are grouped
- No sub-queue may be longer than 4 entries

```
List<QueueEntry> requestQueue;
```

```
List<List<QueueEntry>> requestQueue;
```



Load/Write Queue

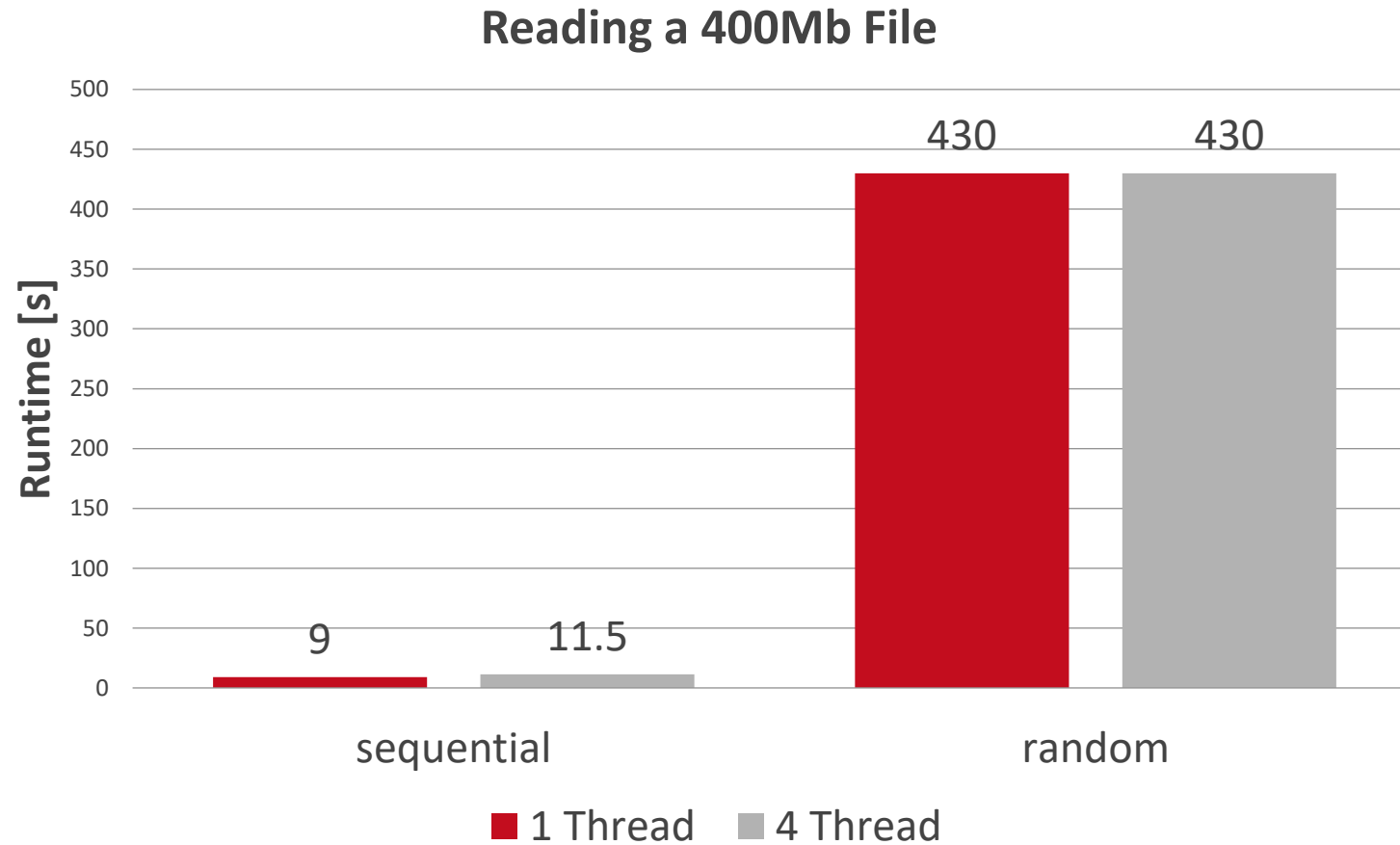
Table 1

3
4
5
6
10
11
13
14

Table 2

1
2
3
4
5
7
14

Command Queuing Mini-Benchmark



Data Structures in the BufferPoolManager to Implement

- Collection of ResourceManagers:
 - There is one for each resource ID, so we recommend using a **HashMap<Integer, ResourceManager>**.
- Collection of caches:
 - There is one cache for every PageSize, so we recommend using a **HashMap<PageSize, PageCache>**.
 - When a new ResourceManager is registered, you need to make sure that there is a cache for its PageSize. If there is no Cache for that PageSize, you must **create a new one and add it** to your collection.

Data Structures in the BufferPoolManager to Implement

- Collection of free I/O buffers:
 - There is a collection of buffers (byte []) for every PageSize, so we recommend using a **HashMap<PageSize, List<byte[]>>**.
 - Similar to the collection of caches, when a new ResourceManager is registered, you must ensure a list of free buffers for its PageSize. If there is no list of free buffers for that PageSize, you must **create a new one and add it** to your collection.
 - **BUT** don't forget to **also create the buffers!** After creating the list, you need to create **config.getNumIOBuffers()** empty buffers, where each buffer has the length of the number of bytes requires to store a page, i.e., **byte[pageSize.getNumberOfBytes()]**.

Data Structures in the BufferPoolManager to Implement

- In the core API you will find two classes we recommend using for the request queues **LoadQueueEntry** and **WriteQueueEntry**.
- A load queue:
 - We recommend using a list of lists containing LoadQueueEntries (**List<List<LoadQueueEntry>>**)
 - Make sure to limit the size of each request to a maximum of **MAX_PAGE_REQUESTS_IN_SINGLE_QUEUE** pages.
- A write queue:
 - We recommend using a list of lists containing WriteQueueEntries (**List<List<WriteQueueEntry>>**)
 - When you add a new page into a cache, an old page may get evicted. You must create a **WriteQueueEntry** with that evicted page and then add it to your write queue **only if it has been modified.**
 - If it wasn't modified, there is no need to write, so just add the buffer back to the list of free I/O buffers.

Threading and Synchronization

Synchronization and Consistency

- Multiple Query Threads access the same data structures, e.g., caches, I/O Queues.
- Make sure that the **concurrent access/modification is supported**
 - For example, use concurrent blocking queues or concurrent hash maps (later)
 - Lock structures (later)
- Examples of cases to handle (to avoid):
 - 1) Two read requests to a non-cached page come quickly. Both requests end in the load queue, and two versions of the same page are loaded into the cache.
 - 2) A page is evicted from the cache and in the write queue. Another load request to that page comes directly afterward and ends in the load queue.
The load request is handled before the write request: The old unmodified page version is re-read, which is no longer valid.

Example Critical Situation 1: Two Read Requests for the Same Page



Execution Thread 1: **read**(P)

Execution Thread 2: **read**(P)

Cache

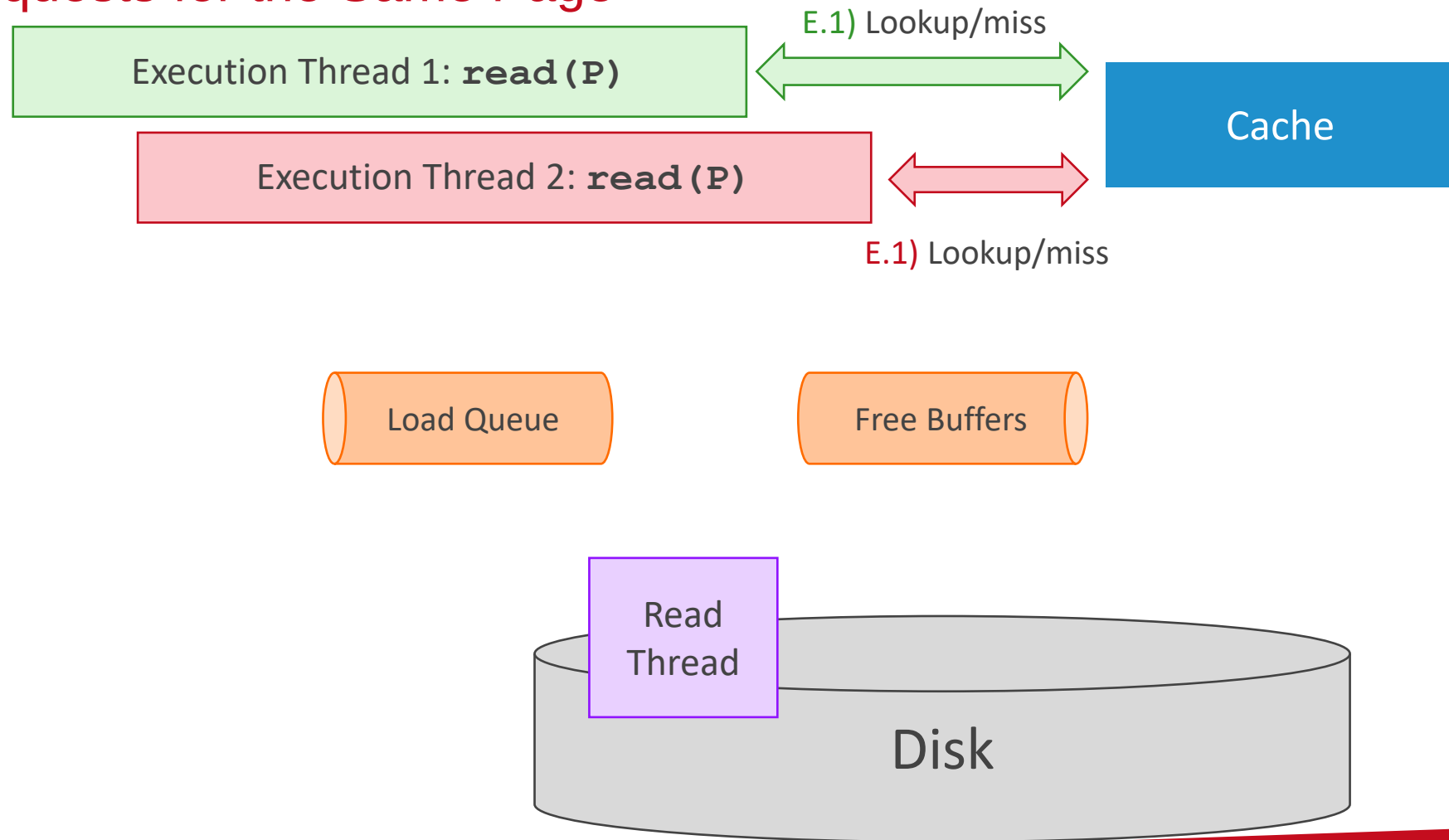
Load Queue

Free Buffers

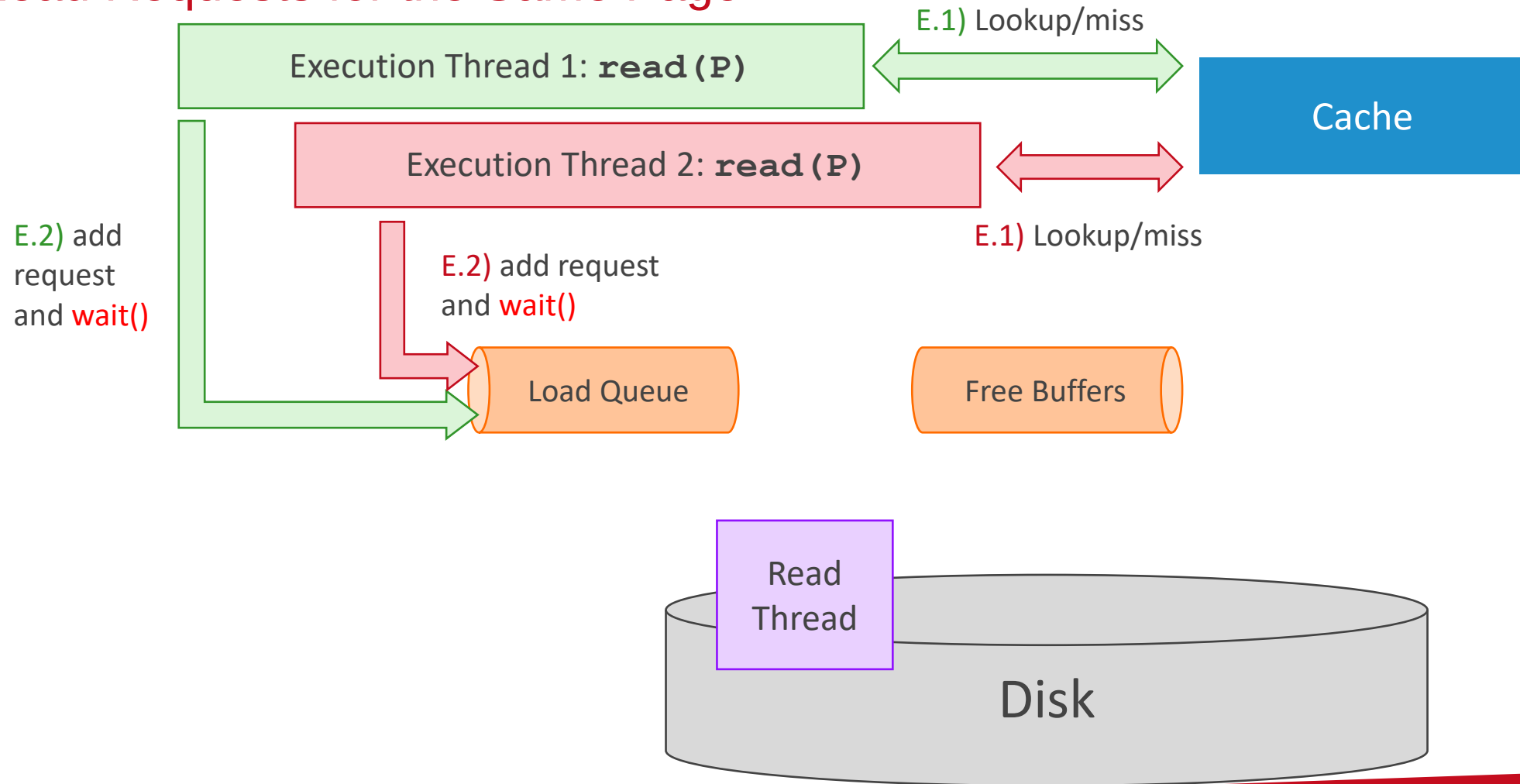
Read
Thread

Disk

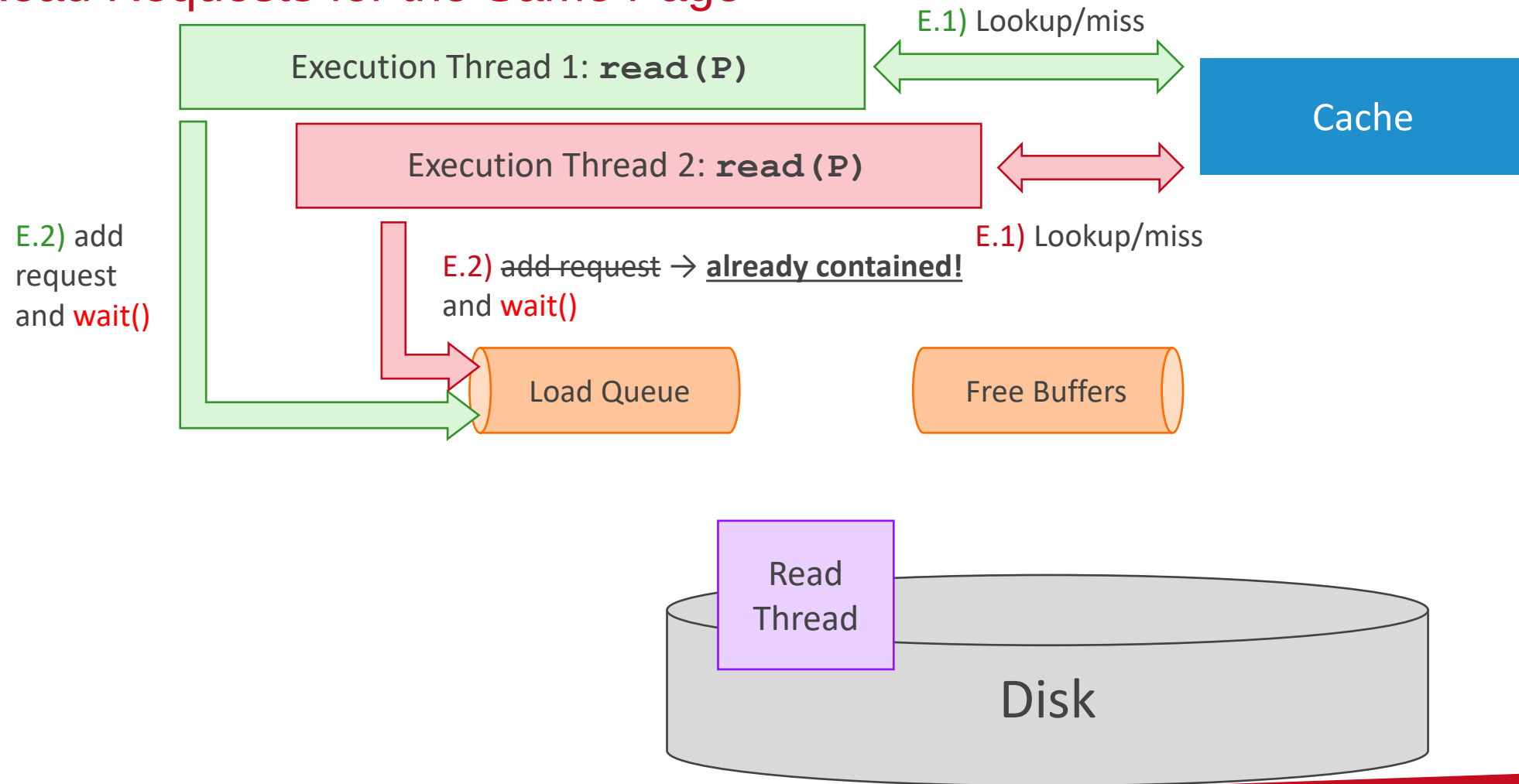
Example Critical Situation 1: Two Read Requests for the Same Page



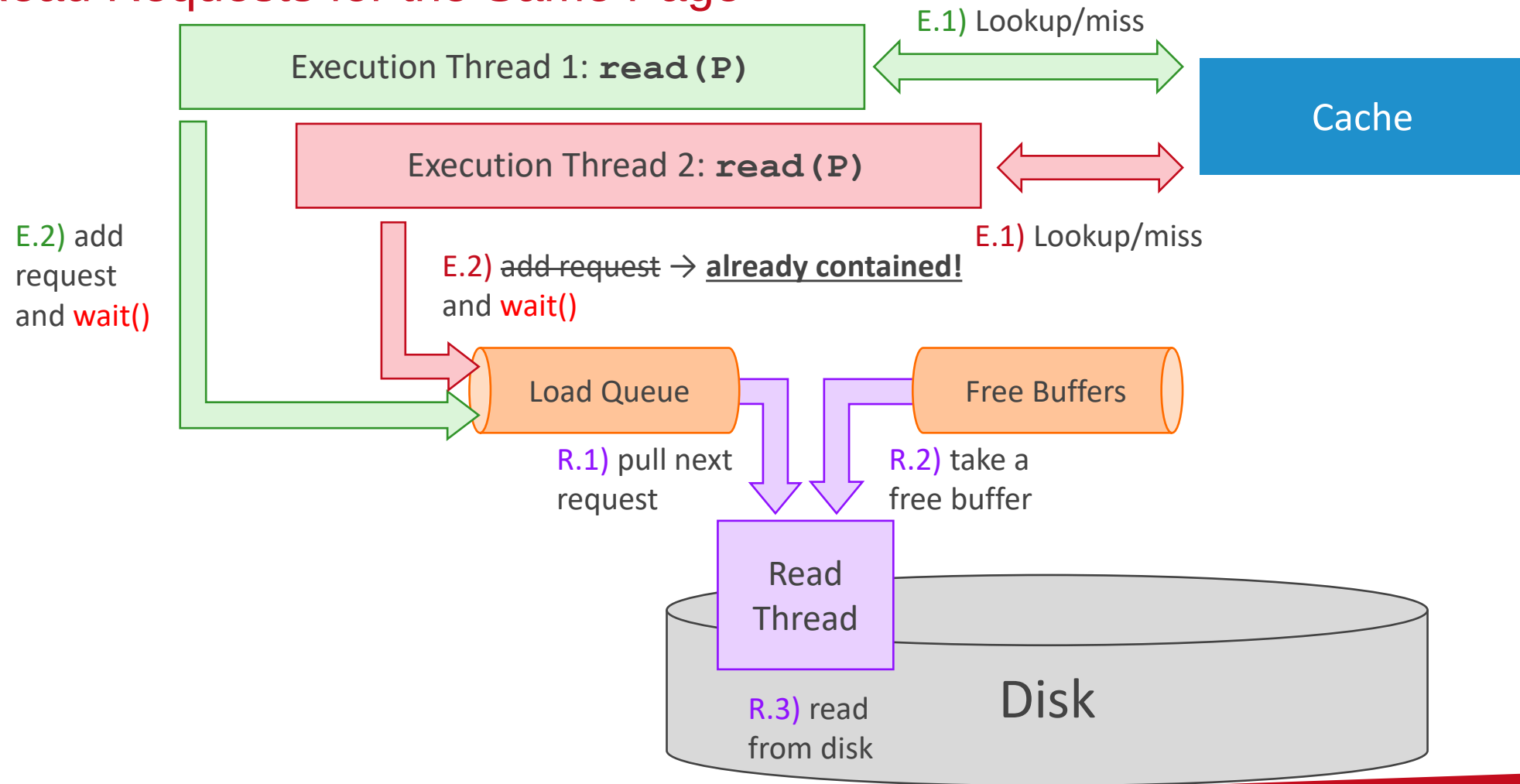
Example Critical Situation 1: Two Read Requests for the Same Page



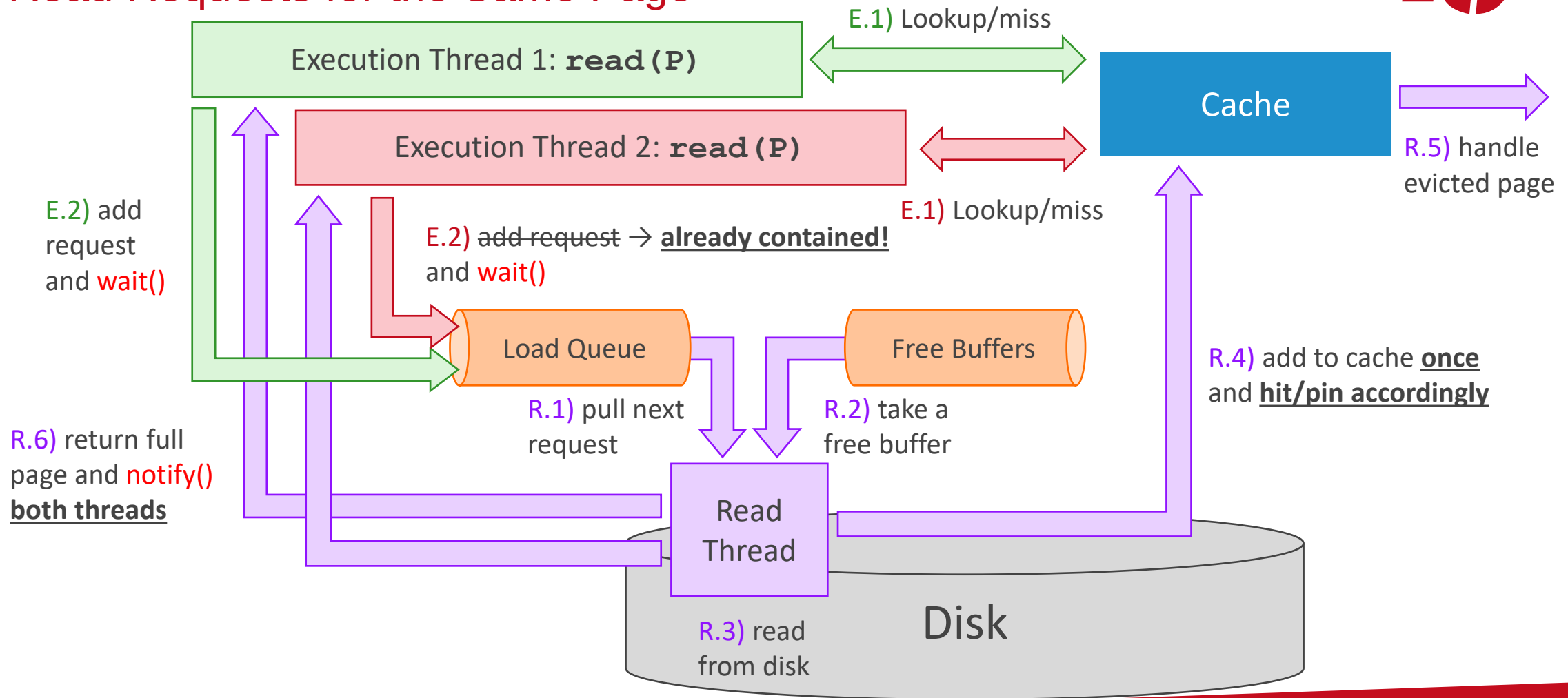
Example Critical Situation 1: Two Read Requests for the Same Page



Example Critical Situation 1: Two Read Requests for the Same Page



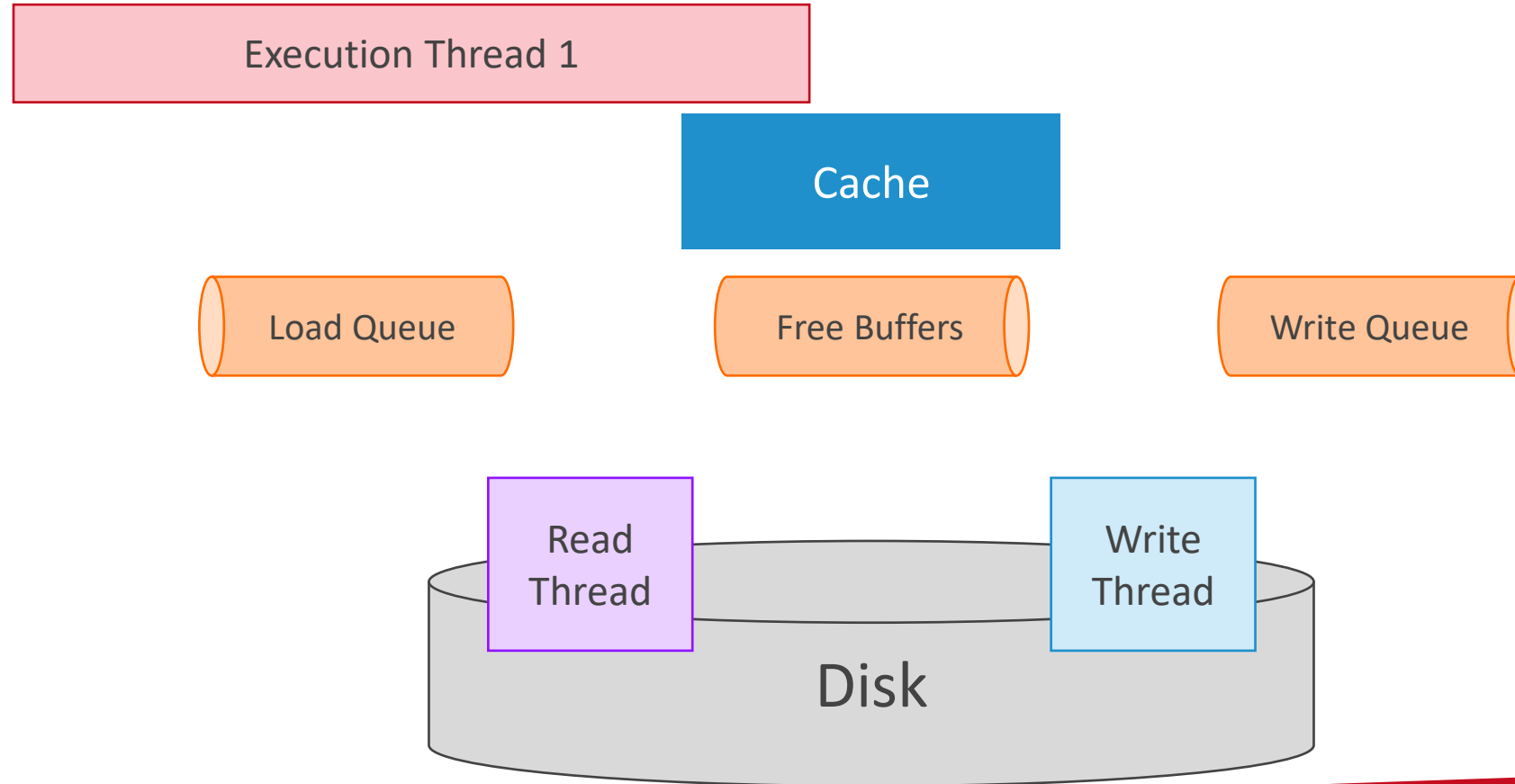
Example Critical Situation 1: Two Read Requests for the Same Page



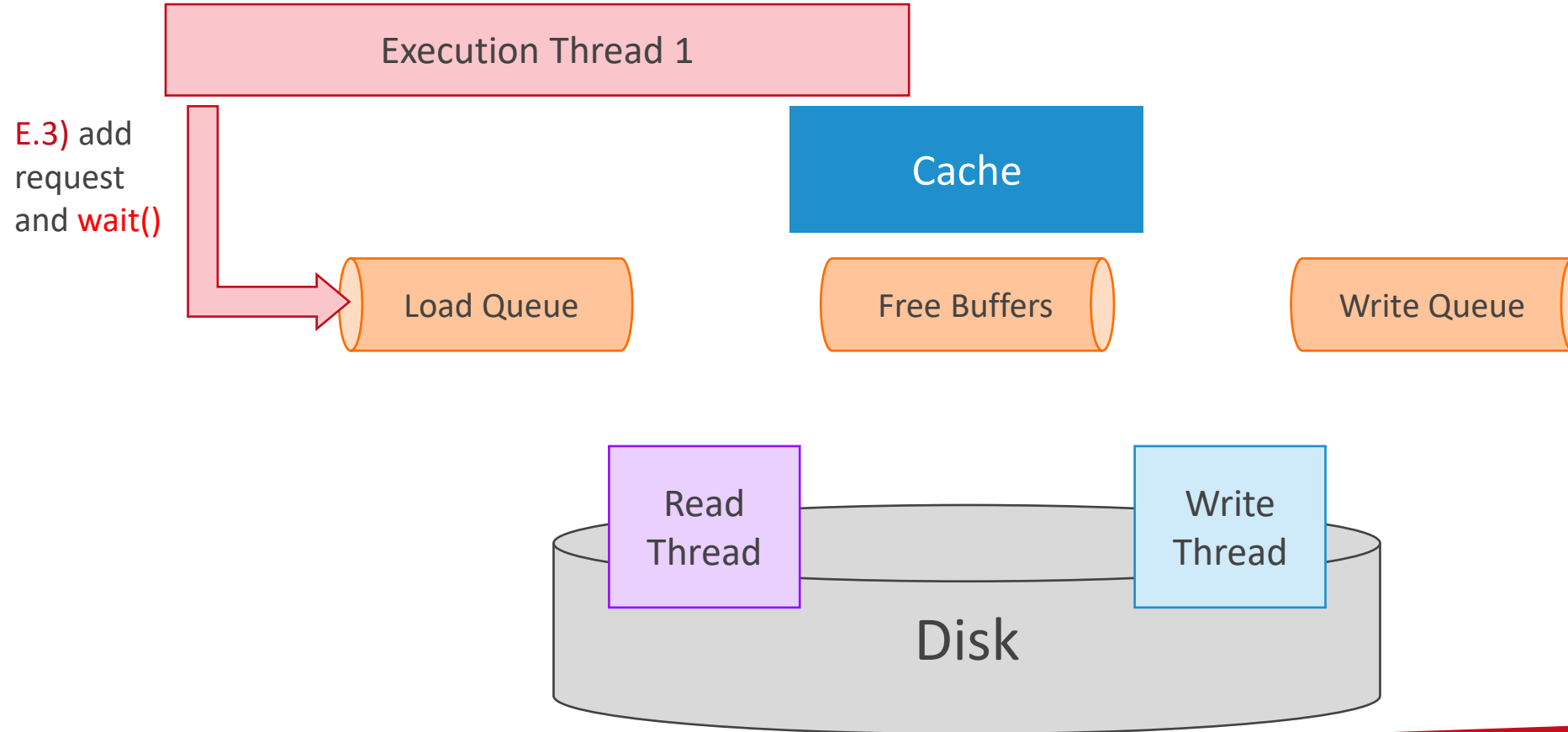
Example critical situation: Two read requests for the same page

- Both threads issue a request for the same page
- **Only one request** ends up in the request queue
- Upon finishing the reading, the page needs to be returned to the threads and
 - Enter the cache
 - Receive two hits
 - Receive two pin() requests!
- Solution:
 - First adds the requested page to the cache (addAndPin()), and then call getAndPin().
 - Both operations account for one hit and one pin.

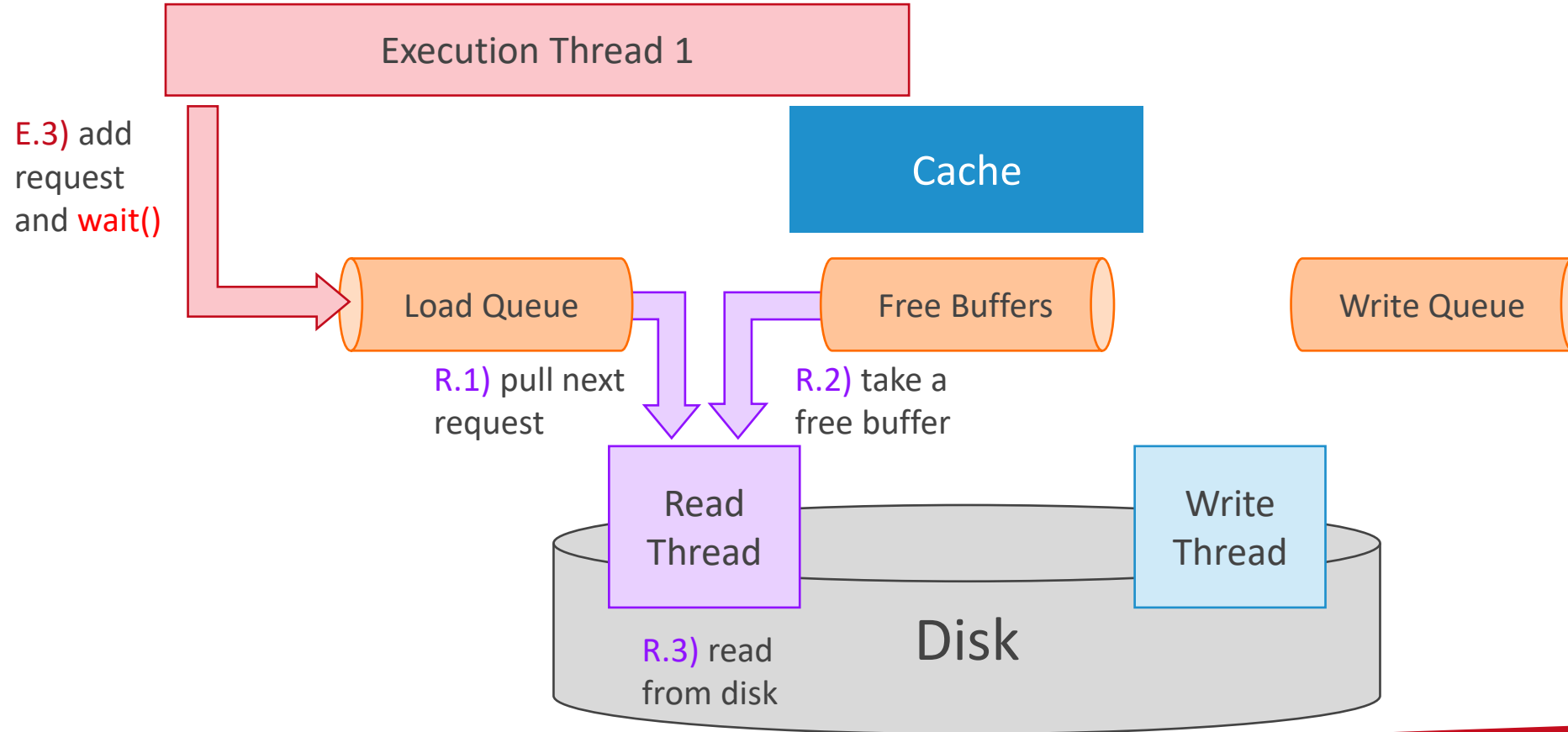
Example Critical Situation 2: Read Request Processed Before the New Value is Written



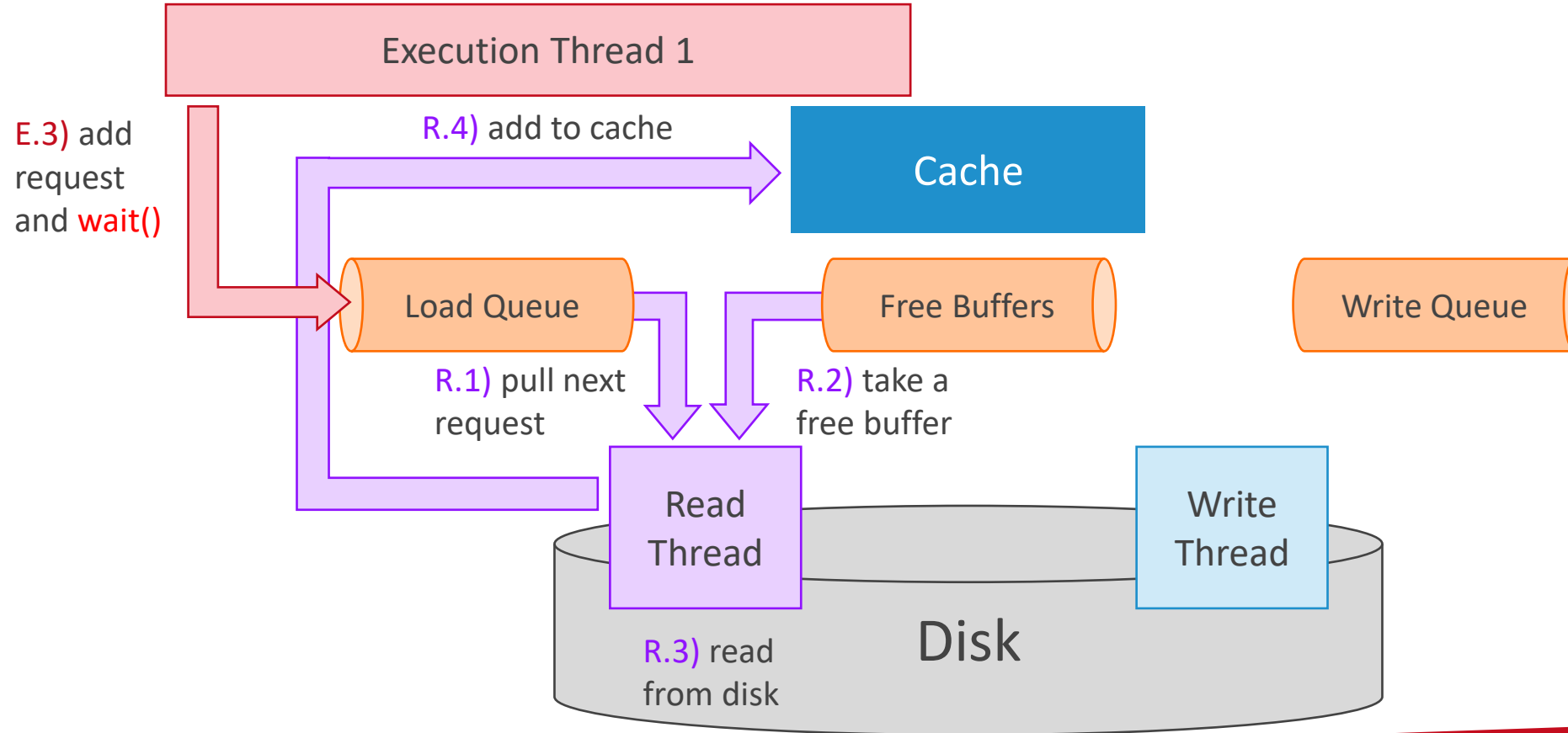
Example Critical Situation 2: Read Request Processed Before the New Value is Written



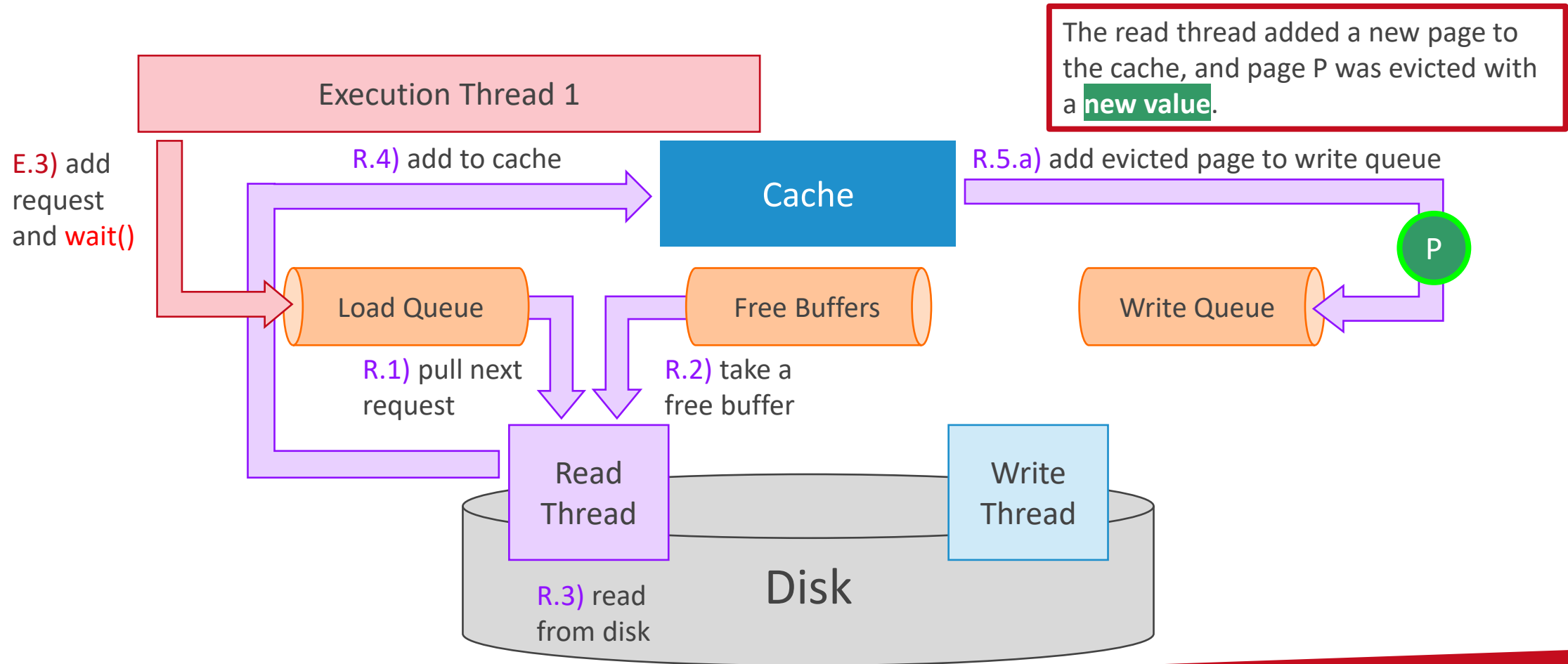
Example Critical Situation 2: Read Request Processed Before the New Value is Written



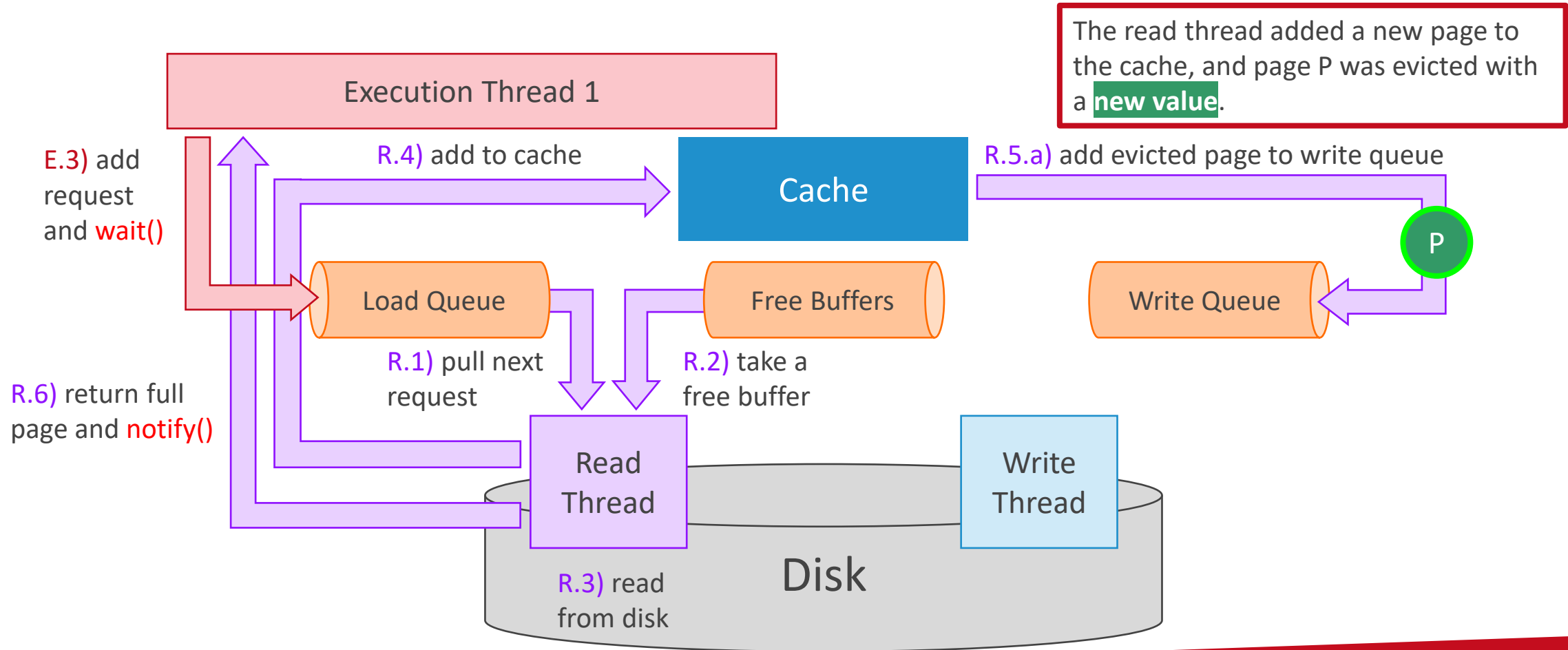
Example Critical Situation 2: Read Request Processed Before the New Value is Written



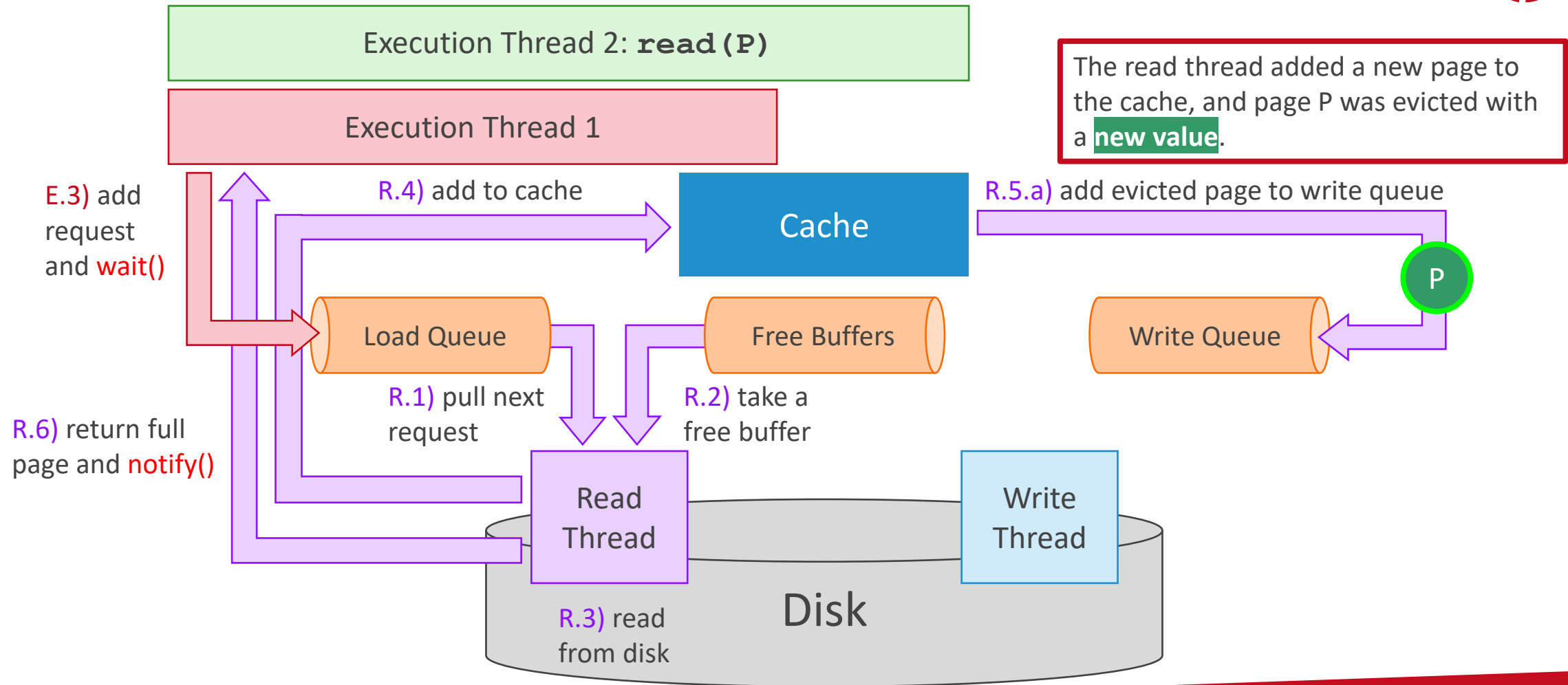
Example Critical Situation 2: Read Request Processed Before the New Value is Written



Example Critical Situation 2: Read Request Processed Before the New Value is Written



Example Critical Situation 2: Read Request Processed Before the New Value is Written

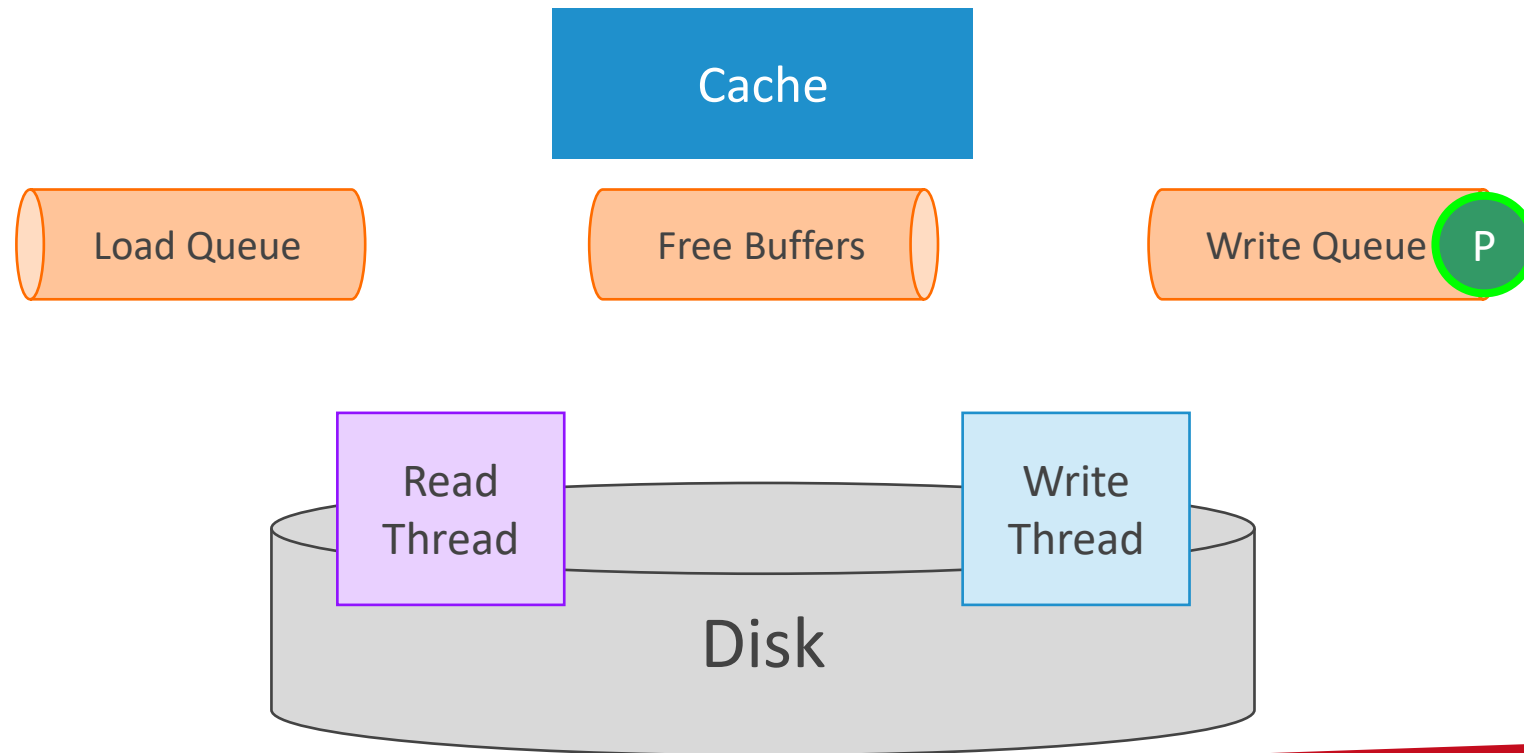


Example Critical Situation 2: Read Request Processed Before the New Value is Written

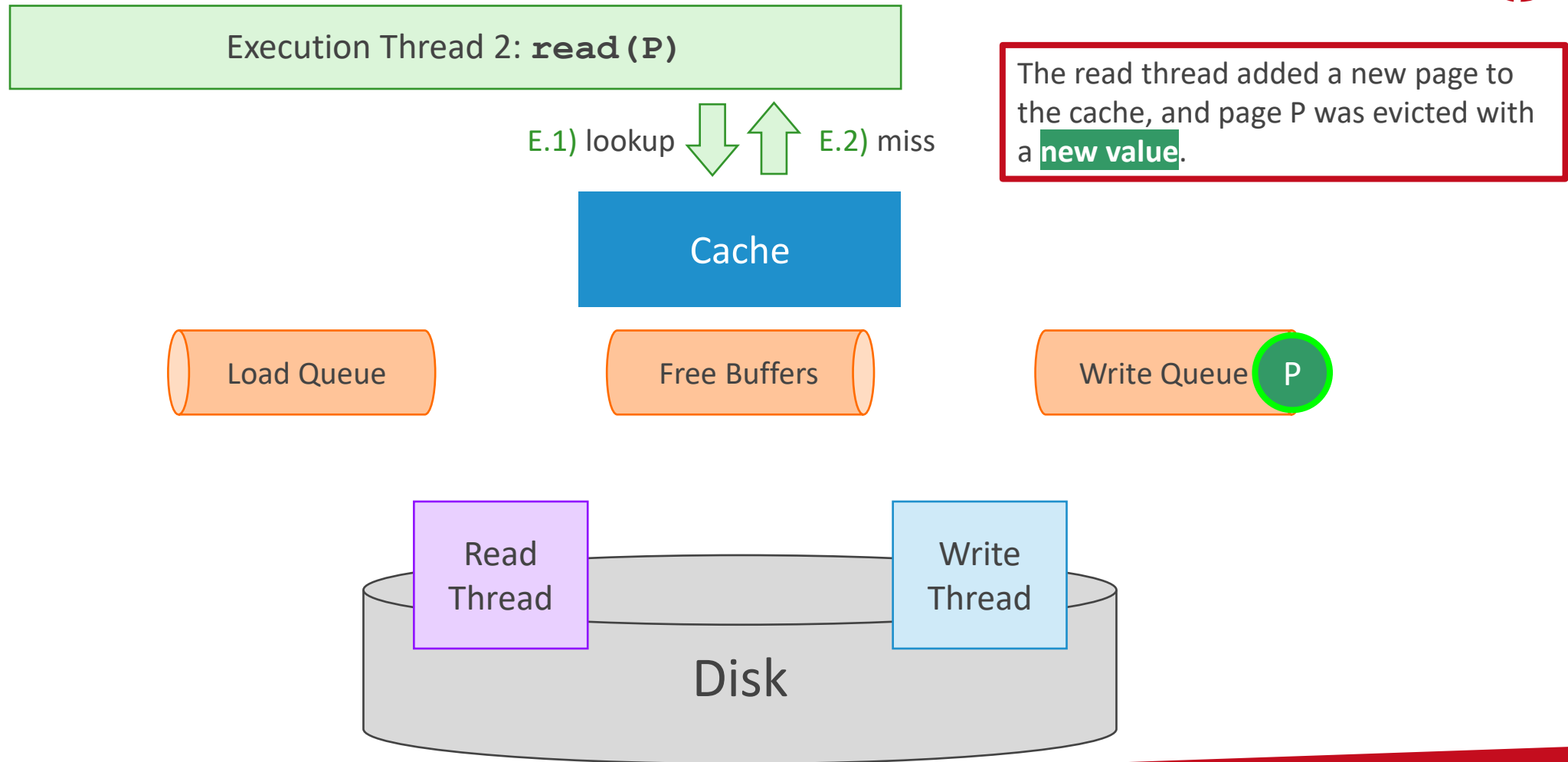


Execution Thread 2: **read (P)**

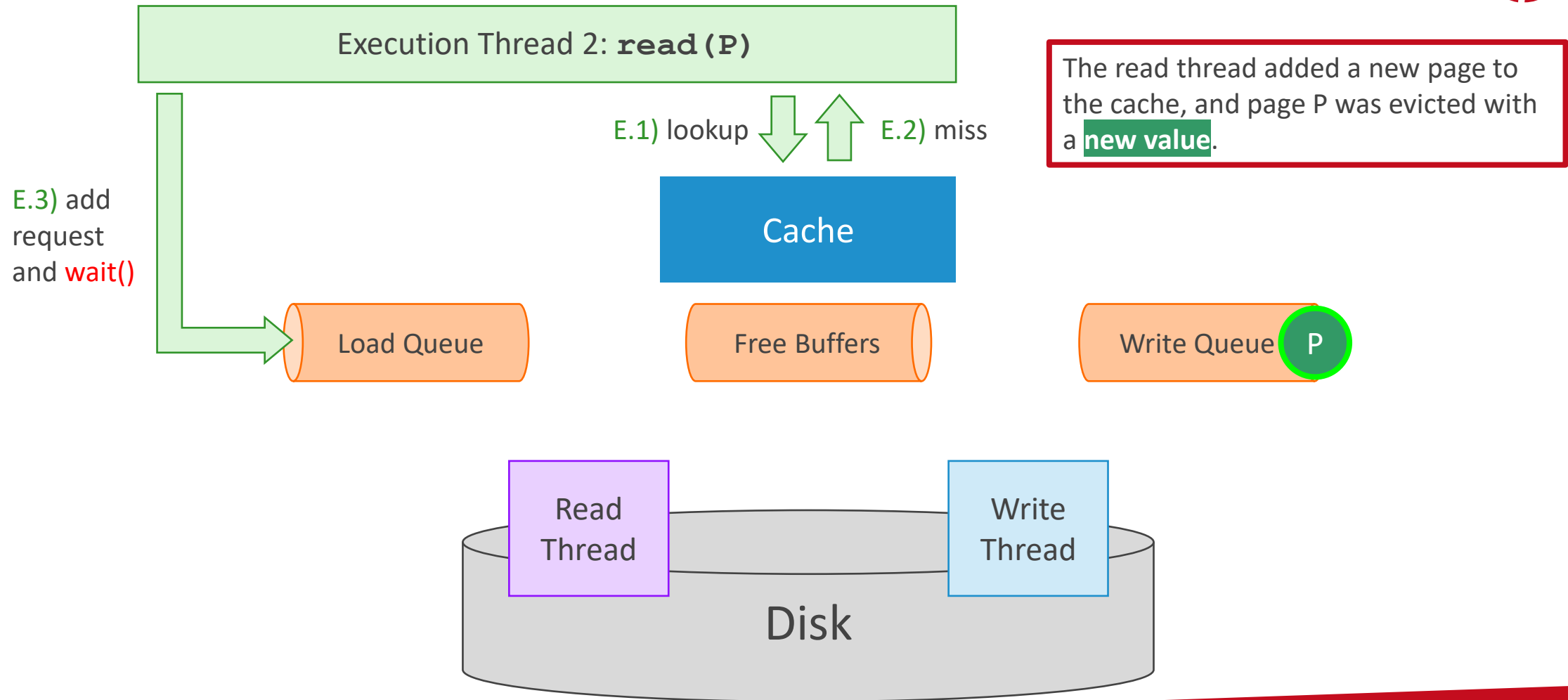
The read thread added a new page to the cache, and page P was evicted with a **new value**.



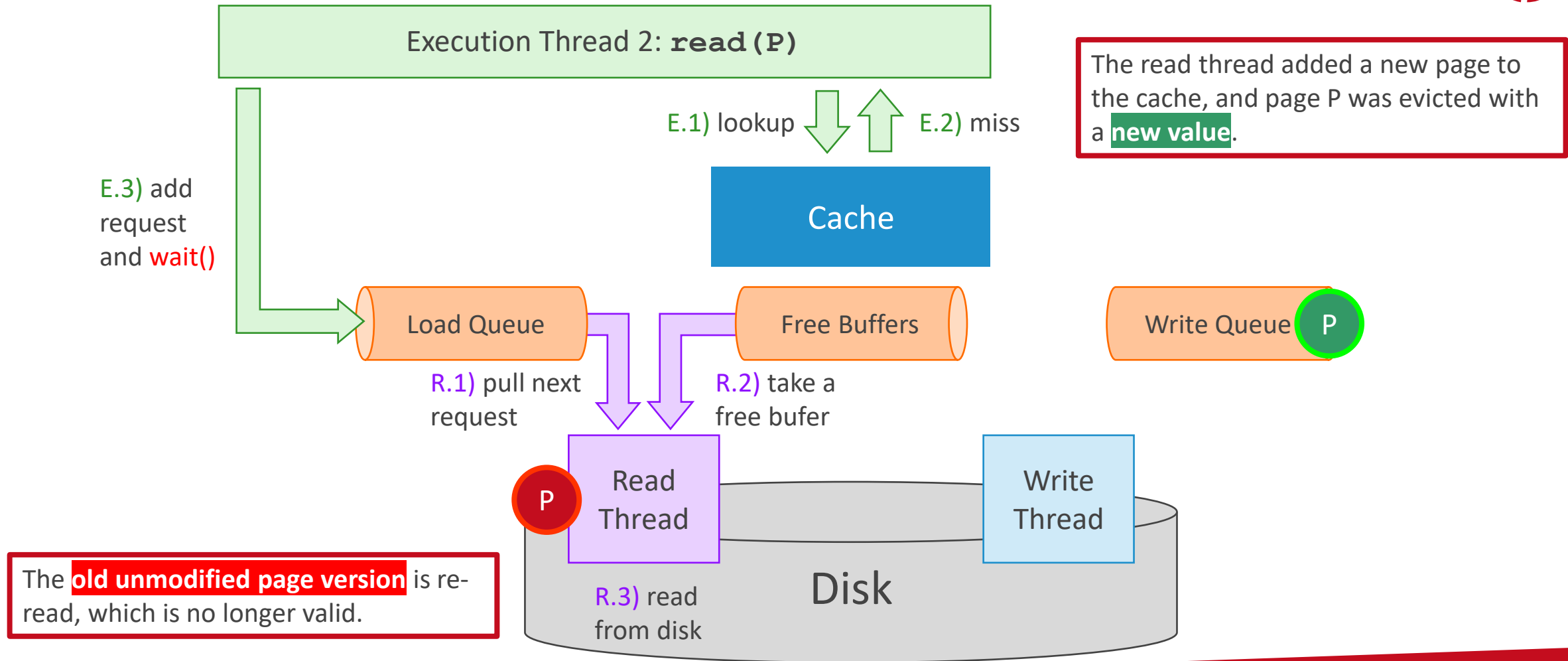
Example Critical Situation 2: Read Request Processed Before the New Value is Written



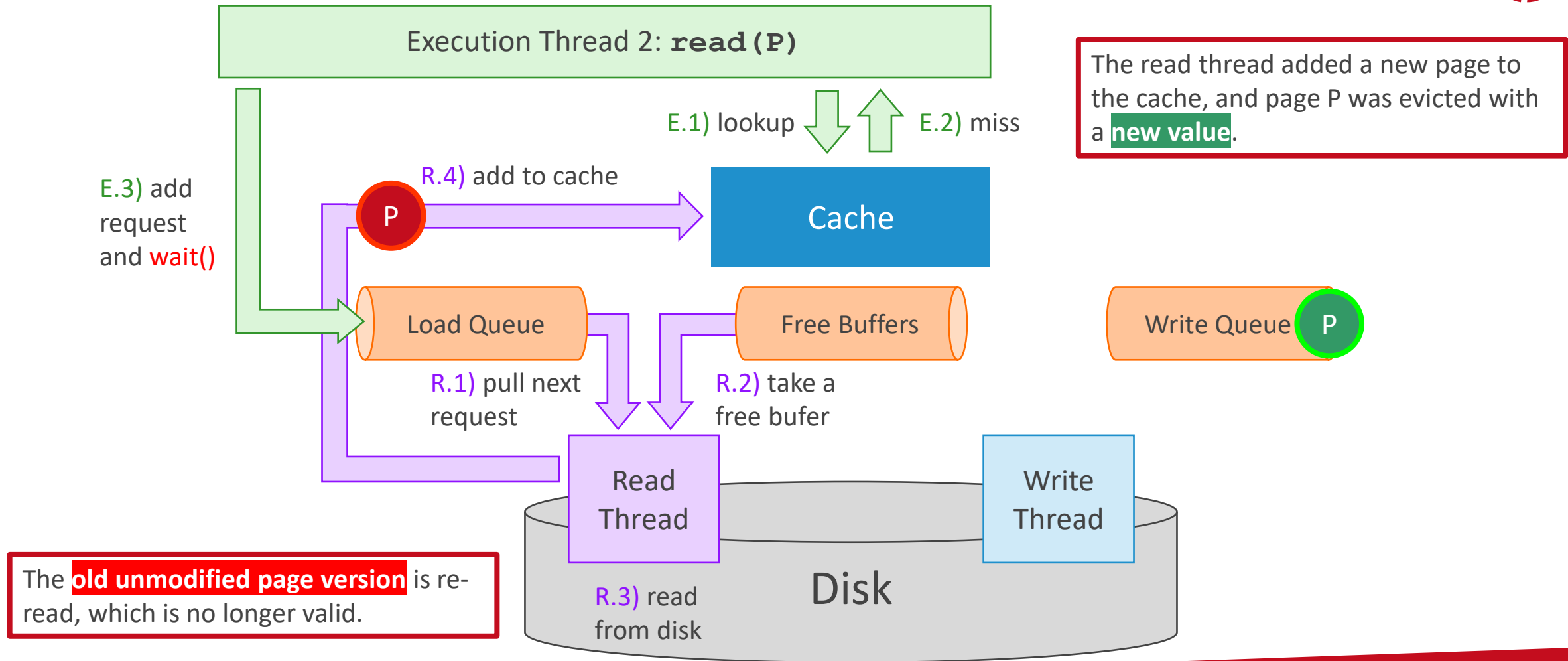
Example Critical Situation 2: Read Request Processed Before the New Value is Written



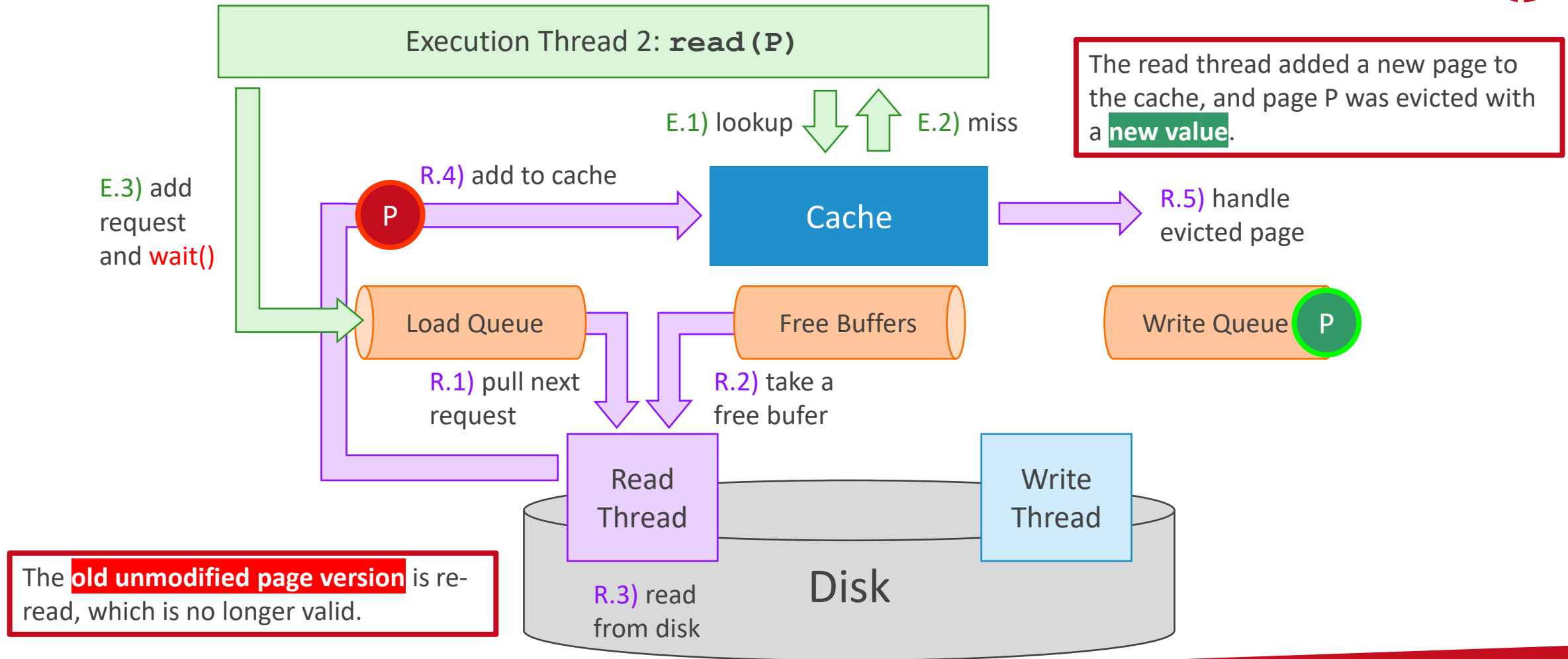
Example Critical Situation 2: Read Request Processed Before the New Value is Written



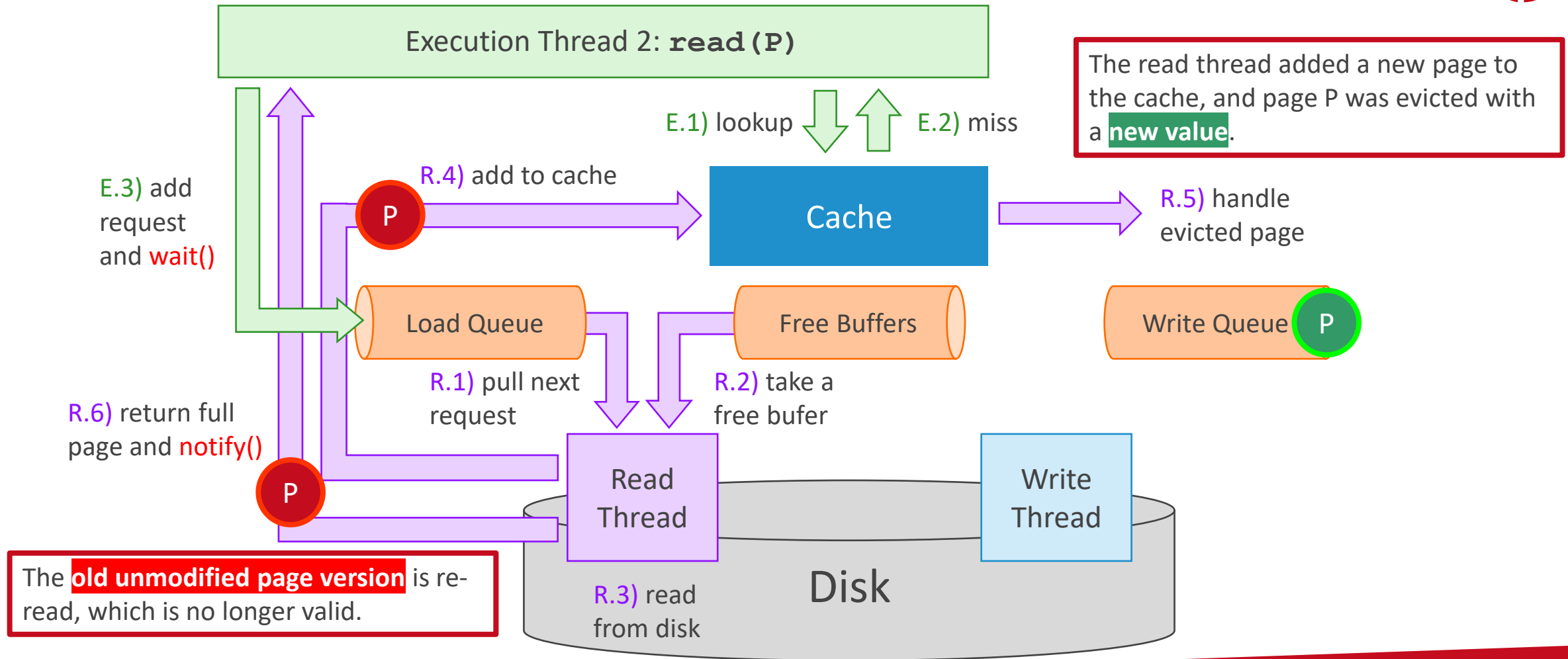
Example Critical Situation 2: Read Request Processed Before the New Value is Written



Example Critical Situation 2: Read Request Processed Before the New Value is Written



Example Critical Situation 2: Read Request Processed Before the New Value is Written

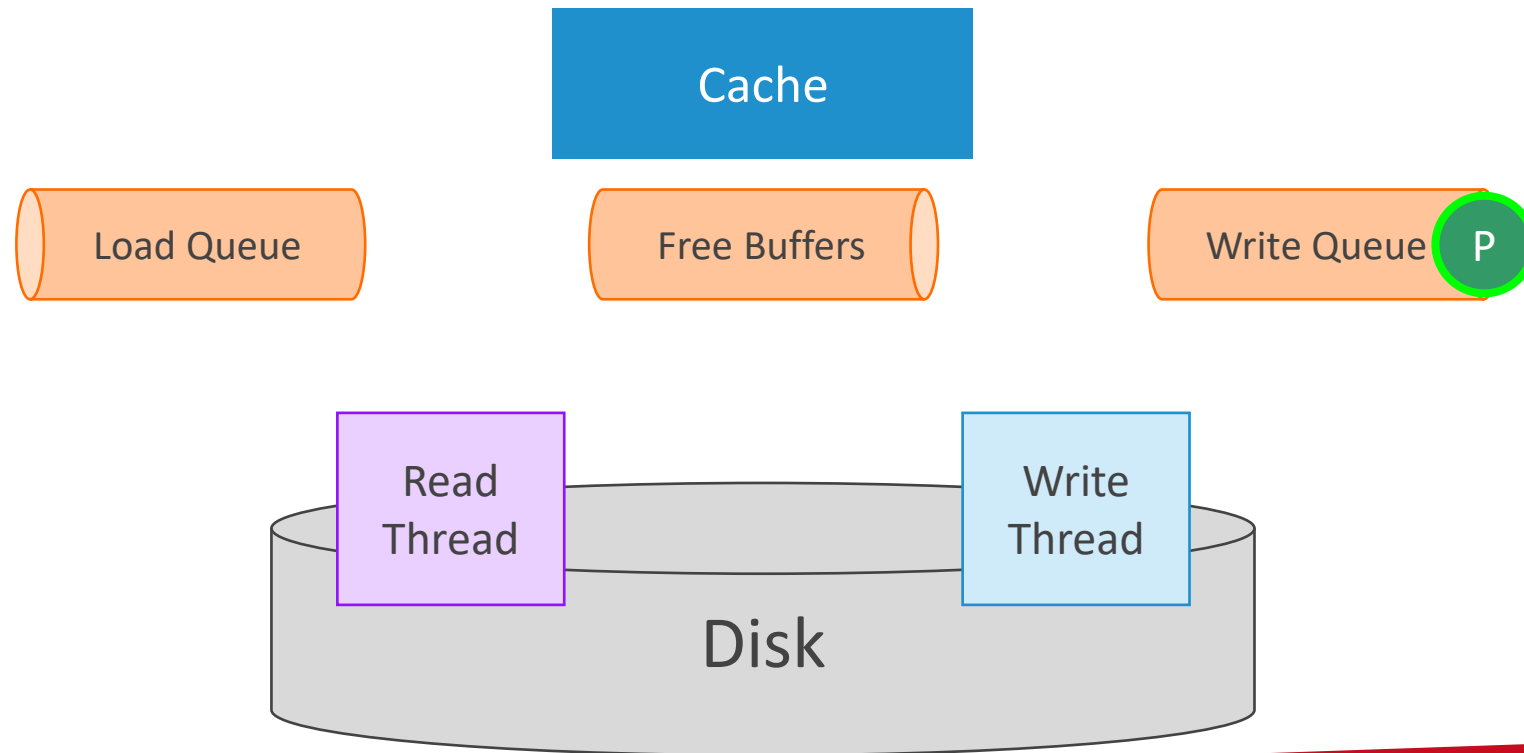


Example Critical Situation 2: Read Request Processed Before the New Value is Written

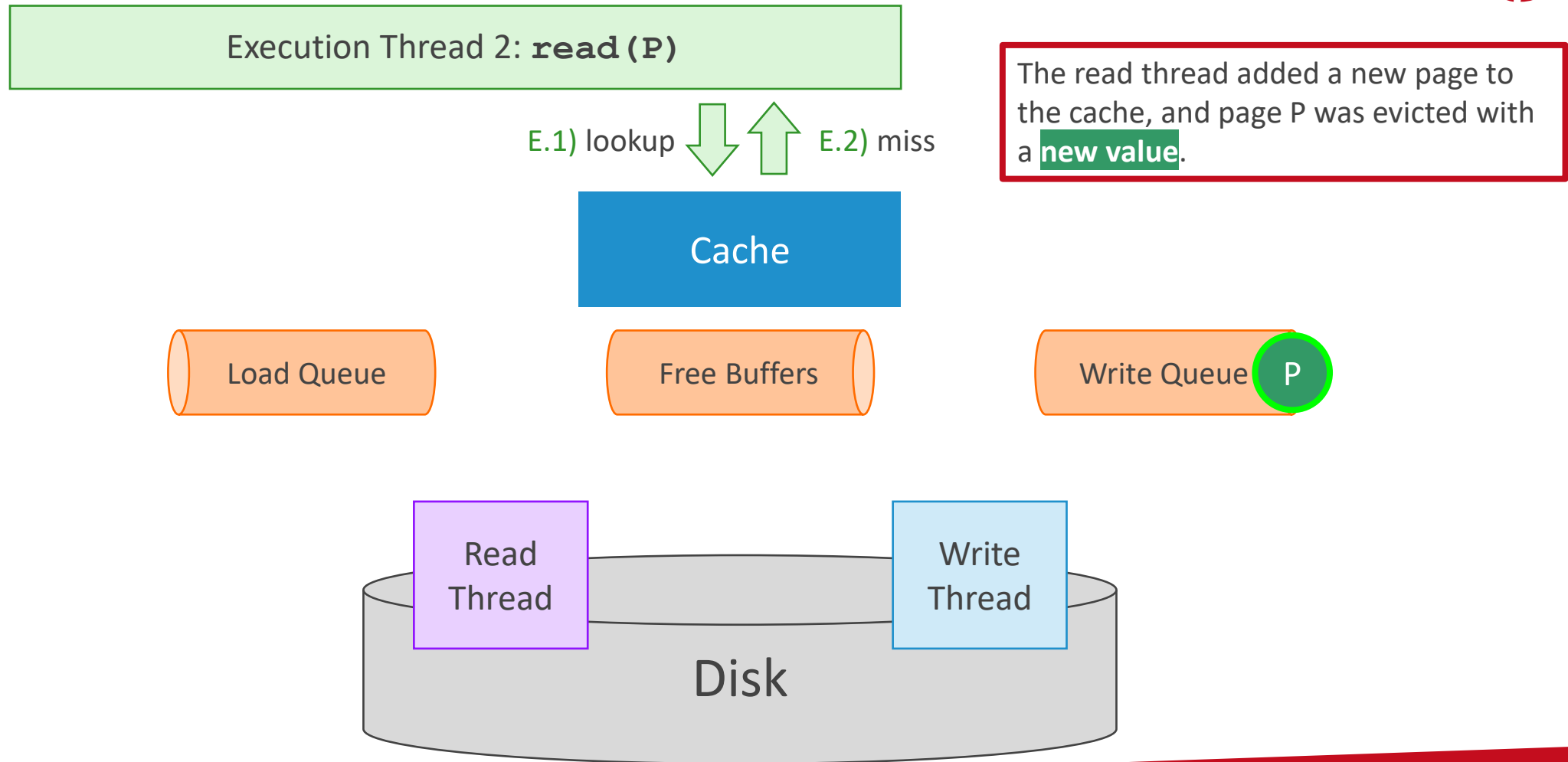


Execution Thread 2: **read (P)**

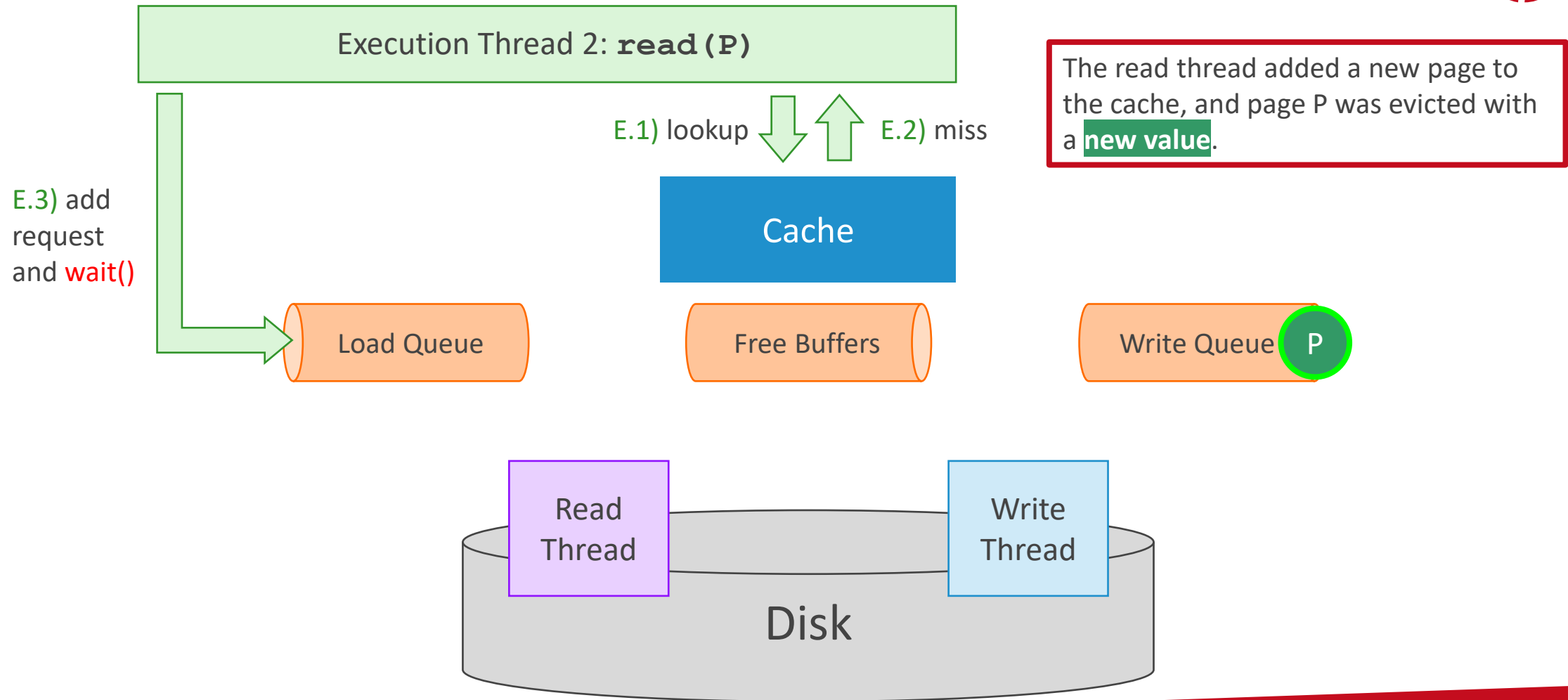
The read thread added a new page to the cache, and page P was evicted with a **new value**.



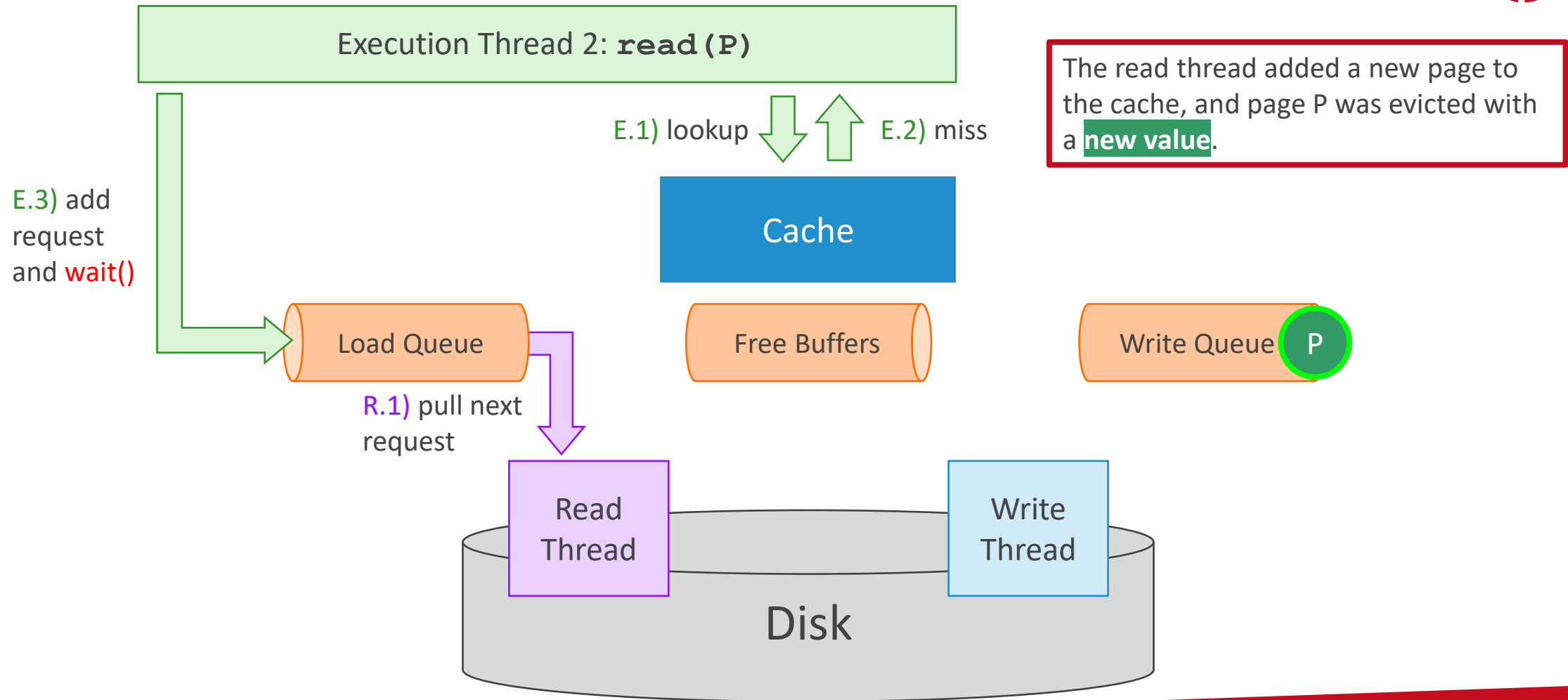
Example Critical Situation 2: Read Request Processed Before the New Value is Written



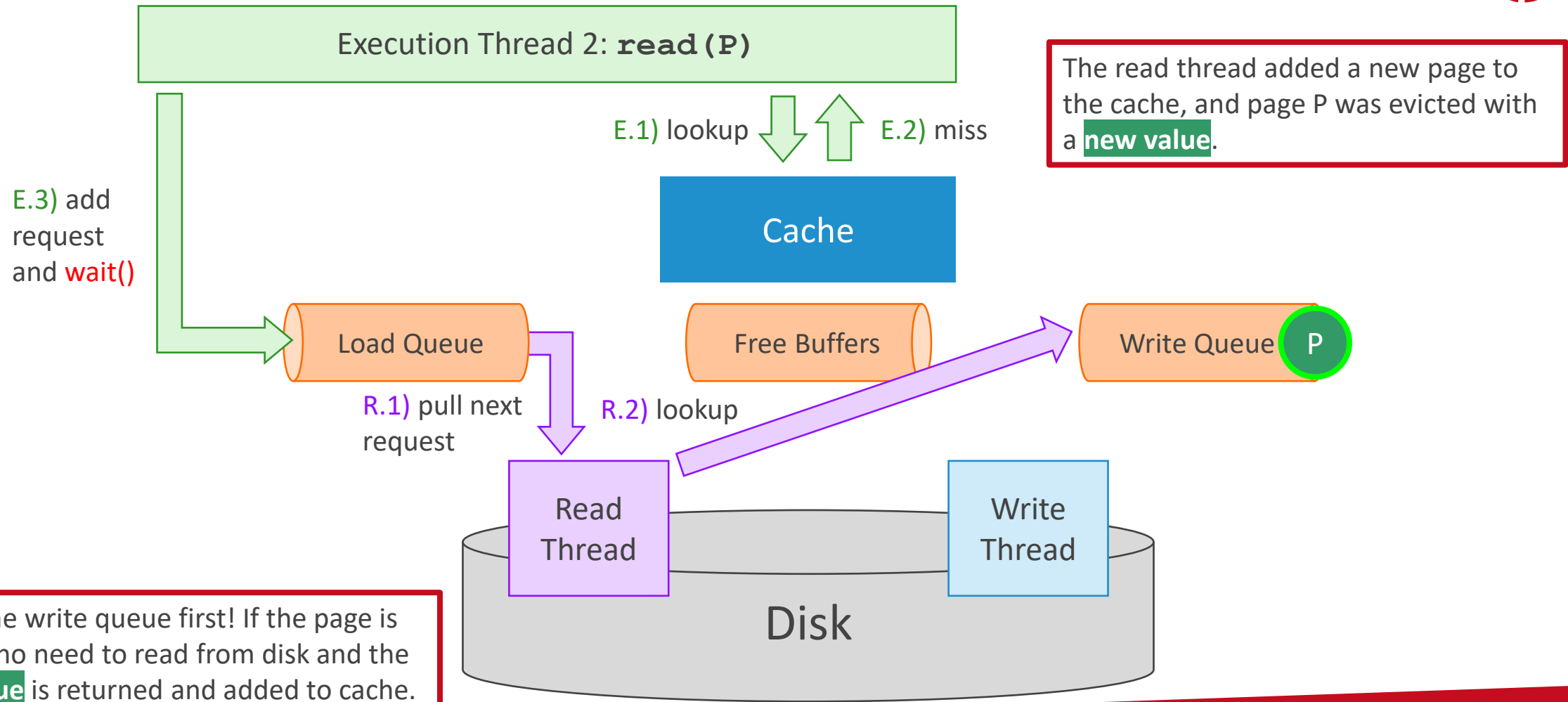
Example Critical Situation 2: Read Request Processed Before the New Value is Written



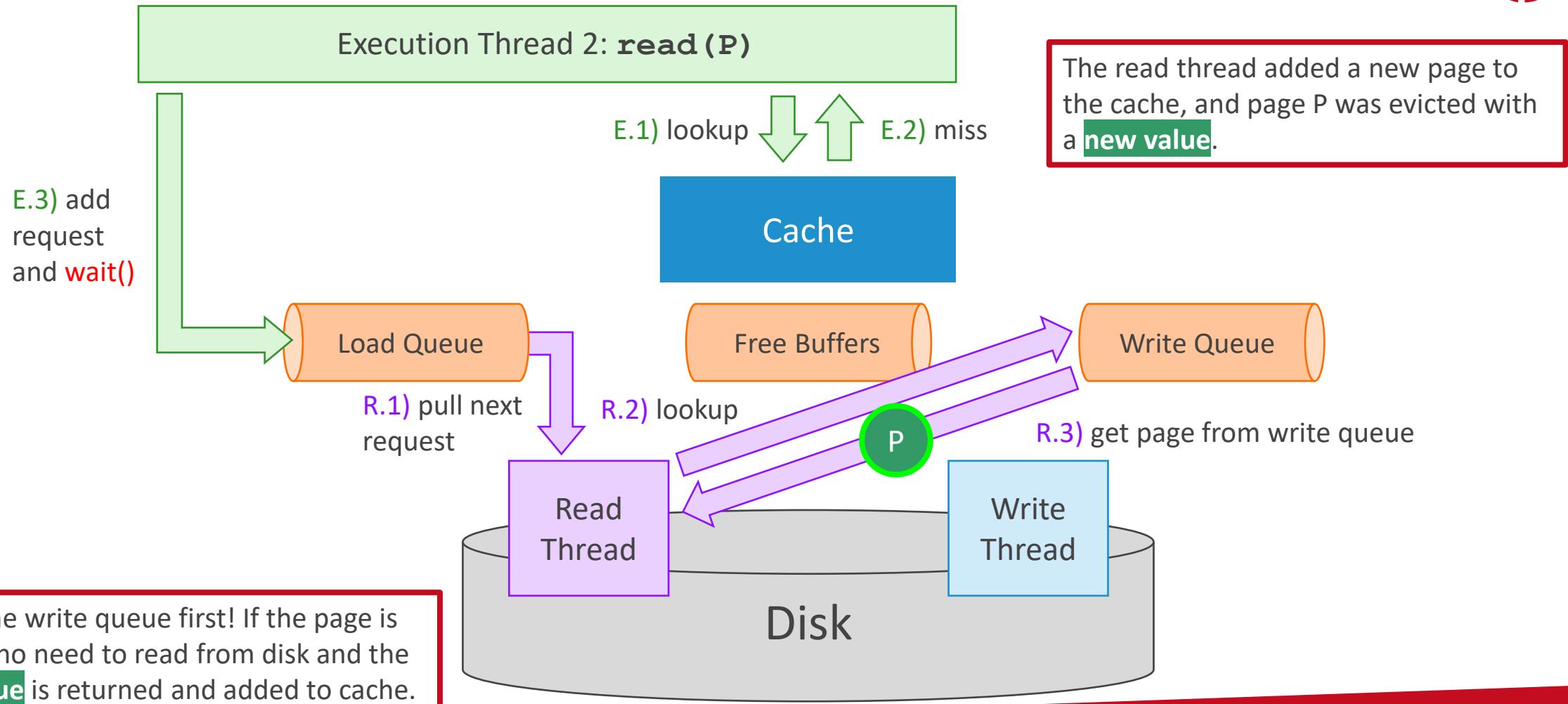
Example Critical Situation 2: Read Request Processed Before the New Value is Written



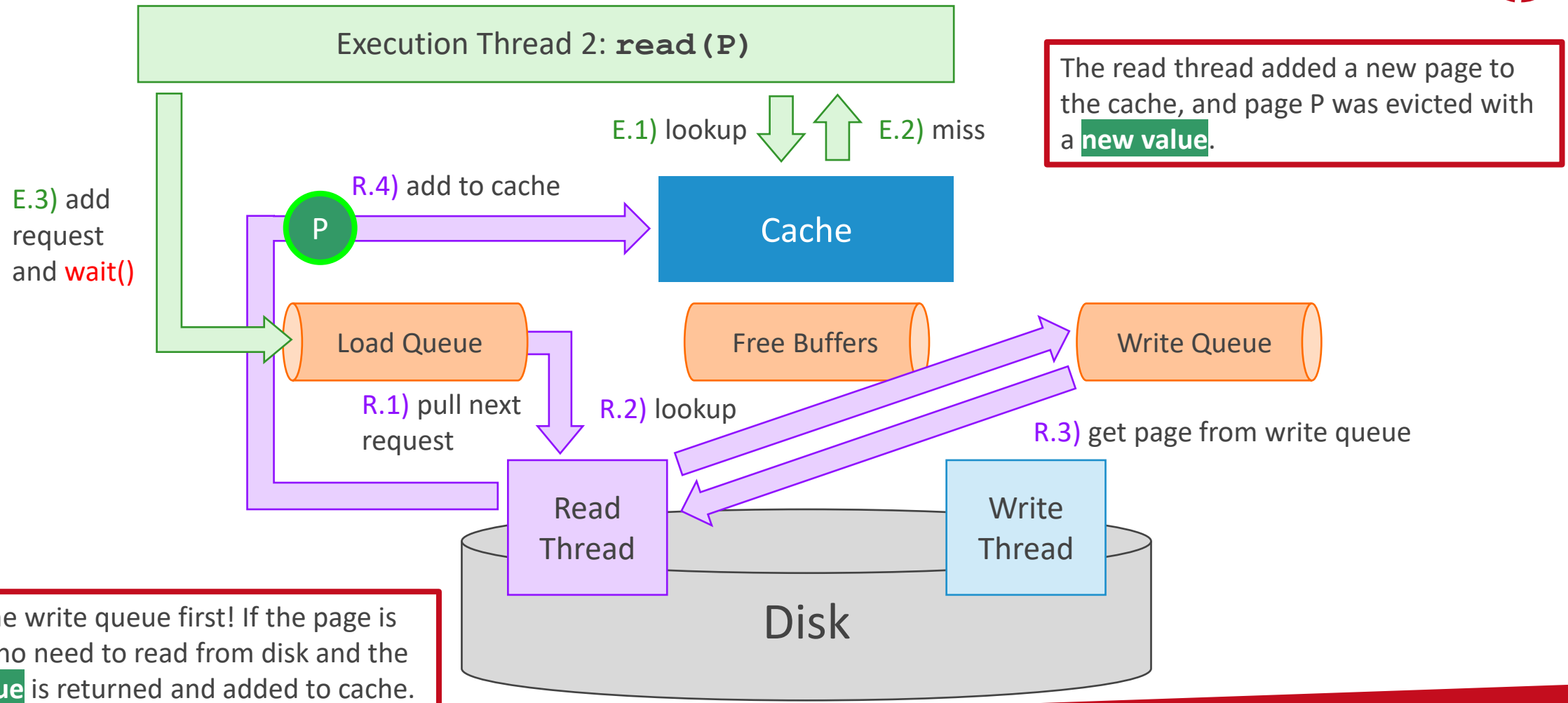
Example Critical Situation 2: Read Request Processed Before the New Value is Written



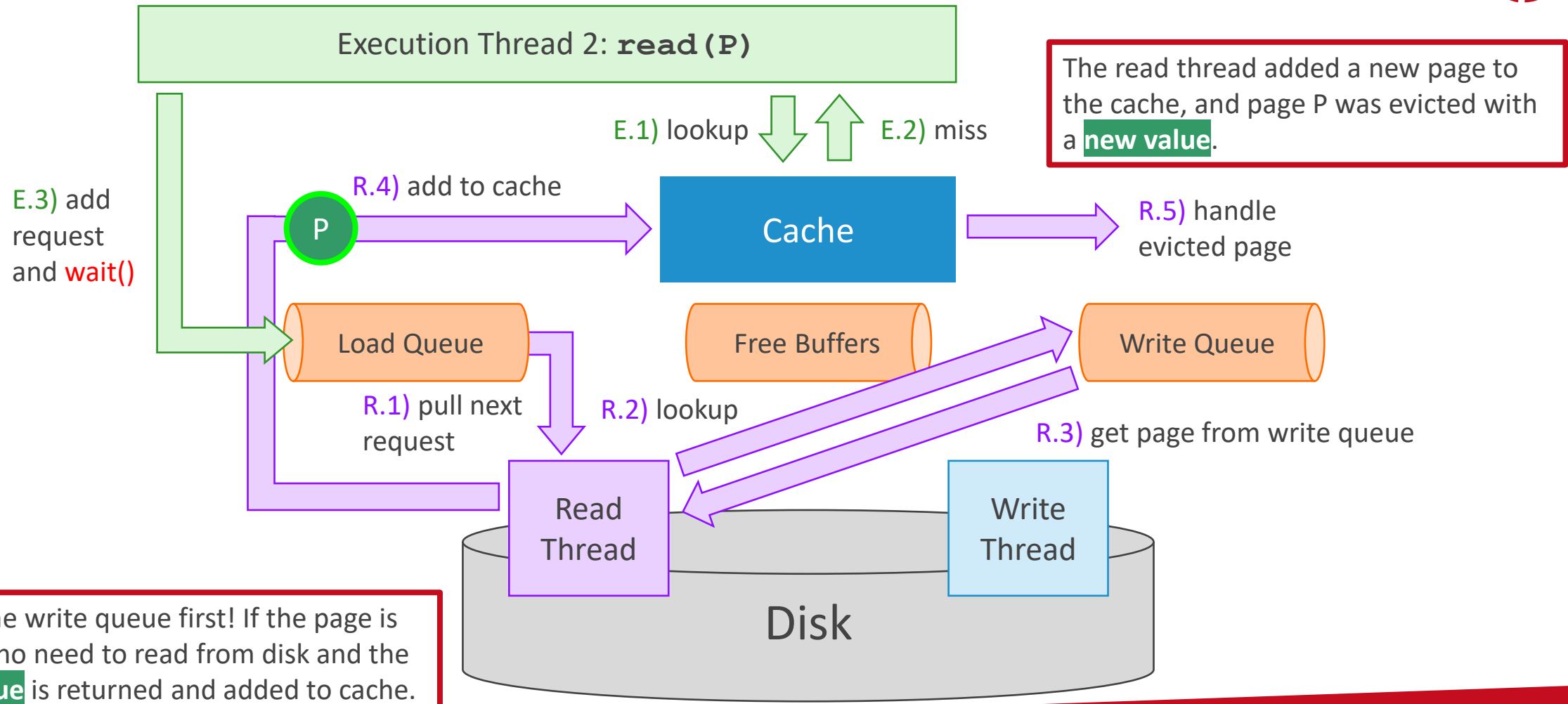
Example Critical Situation 2: Read Request Processed Before the New Value is Written



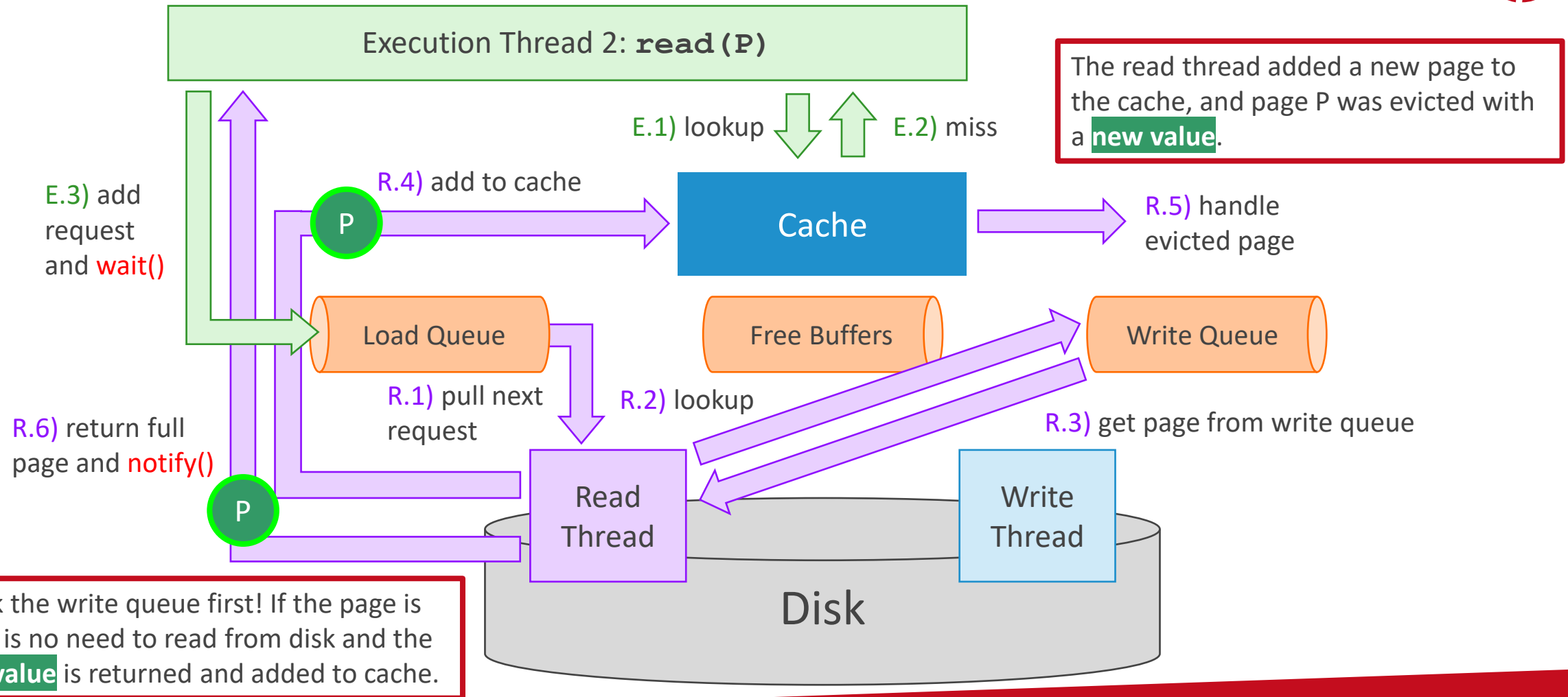
Example Critical Situation 2: Read Request Processed Before the New Value is Written



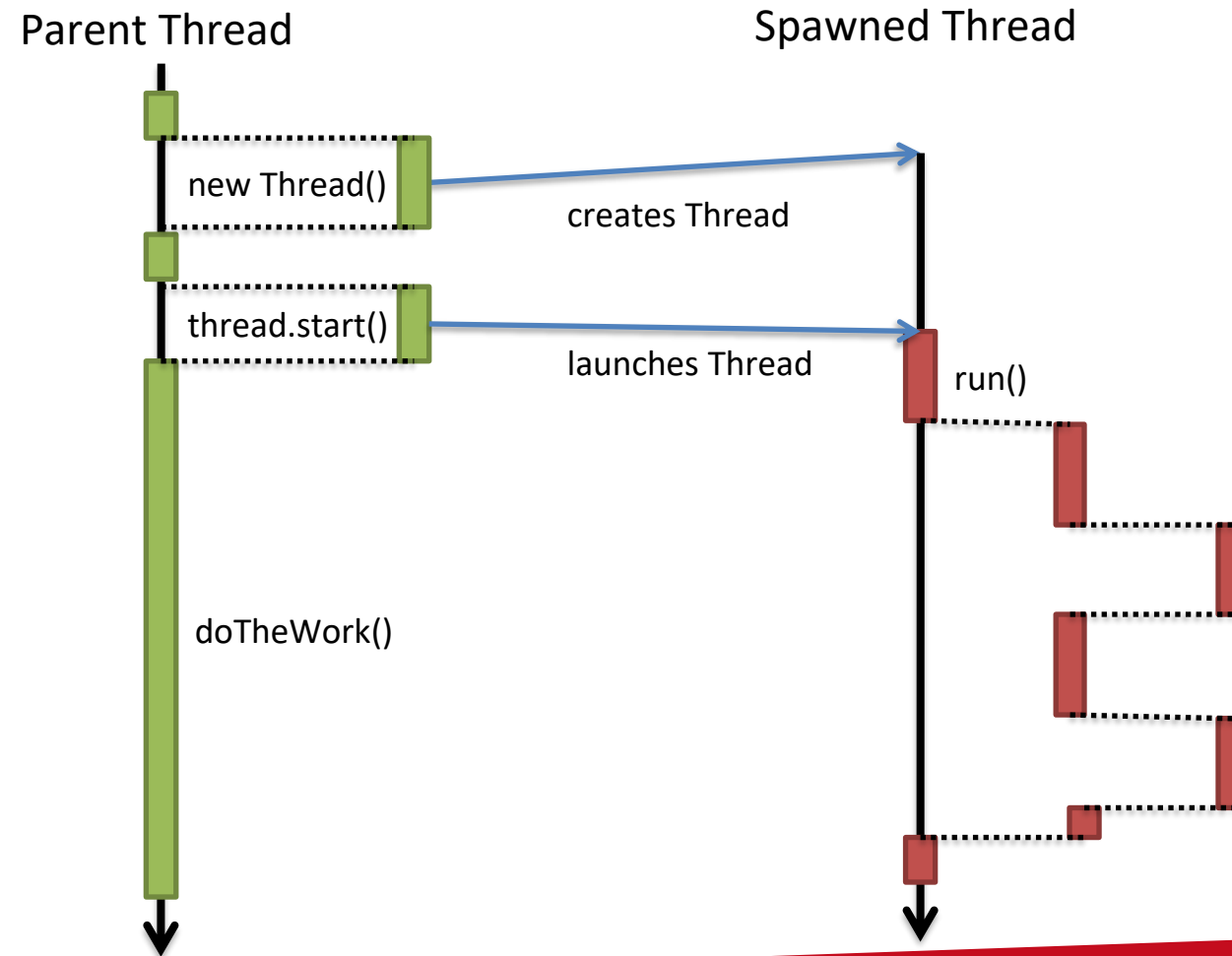
Example Critical Situation 2: Read Request Processed Before the New Value is Written



Example Critical Situation 2: Read Request Processed Before the New Value is Written



Creating and Ending Threads in Java



Threading Code Example

Reading Thread Code

1.

```
public class ReadThread extends Thread
{
2.   private volatile boolean alive = true;
3.   public void shutdown() {
        this.alive = false;
    }
4.   public void run()
    {
        while (alive) {
            PageReq toRead = queue.pop();
            Page p = readPage(toRead);
            handlePage(p);
        }
    }
}
```

Buffer Pool Code

```
public void startPool()
{
    createStructures();
    createQueues();

    rT = new ReadThread();
    rT.start();
}

public void closeBufferPool()
{
    rt.shutdown();

    destroyQueues();
    destroyStructures();
}
```

Threading Code Example

Reading Thread Code

```
public class ReadThread extends Thread
{
    private volatile boolean alive = true;

    public void shutdown() {
        this.alive = false;
    }

    public void run()
    {
        while (alive) {
            PageReq toRead = queue.pop();
            Page p = readPage(toRead);
            handlePage(p);
        }
    }
}
```

Buffer Pool Code

```
public void startPool()
{
    createStructures();
    createQueues();

    rT = new ReadThread();
    rT.start();
}

public void closeBufferPool()
{
    rt.shutdown();

    destroyQueues();
    destroyStructures();
}
```

1.

2.

3.

Synchronization in Java

- Concurrent (multi threaded) accesses to data structures require some form of synchronization.
- 4 different types of synchronization in Java.
 - 1) Atomic Primitives (package `java.util.concurrent.atomic`).
 - Usable when numeric fields are accessed/updated by multiple threads.
 - Fast but limited in their applicability.
 - 2) Predefined Concurrent Structures (package `java.util.concurrent`).
 - **BlockingQueue**, **ConcurrentMap**.
 - 3) Locks and Conditions (package `java.util.concurrent.locks`).
 - Powerful collection of locks. Supports fair scheduling, lock sharing, and upgrading.
 - 4) Language built-in: "**synchronized**" keyword.
 - Simpler and currently faster than (3), but less powerful.

Language Built-In Synchronization

- Synchronization is done through so-called "monitors"
- If an operation should be exclusive (no intermediate interference from other threads), obtain the monitor (every object can act as a monitor)

```
private Cache cache;  
private Queue requests;  
  
public void checkAndAdd(num) {  
    Page p = cache.get(num)  
    if (p == null) {  
        requests.push(num);  
    }  
}
```

Concurrent access is possible. Two threads can simultaneously check the cache for the same page and add the same request to the queue.

```
private Cache cache;  
private Queue requests;  
  
public void checkAndAdd(num) {  
    synchronized (cache) {  
        Page p = cache.get(num)  
        if (p == null) {  
            requests.push(num);  
        }  
    }  
}
```

Only one thread at a time may be within the synchronized block. The cache object acts as monitor.

Language Built-In Synchronization

```
private Cache cache;
private Queue requests;

public void checkAndAdd(num) {
    synchronized (cache) {
        Page p = cache.get(num)
        if (p == null) {
            requests.push(num);
        }
    }
}

public void addPage(Page p) {
    synchronized (cache) {
        cache.add(p);
    }
}

private void discard() {
    synchronized (requests) {
        requests.clear();
    }
}
```

Mutually exclusive

NOT exclusive

- A synchronized block is exclusive to all other synchronized blocks on the same monitor object

No Busy Waiting - Notifications

- Problem: Sometimes, a thread can only continue after some other thread has done some work.
- Example: The execution thread needs a page that is not in the cache. It cannot continue until the page has been loaded by the read thread.
- Solution: The execution thread waits on a monitor. The read thread notifies the monitor when the page is loaded.

```
public void doRequest(int page)
{
    Request r = new Request(page);
    loadQueue.push(r);
    synchronized (r) {
        while (!r.isDone()) {
            r.wait();
        }
    }
}
```

Sleeps until notification

```
public void readLoop()
{
    while (alive) {
        Request r = reqQueue.pop();
        load(r);
        r.setDone();
        synchronized (r) {
            r.notifyAll();
        }
    }
}
```

Sends Notification

Threads in Exercise 3

- You need to implement a `ReadThread`
 - Checks if the load queue is not empty.
 - Gets the next load request and processes it:
 - Retrieves the necessary amount of free buffers to complete the request, i.e., if the request asks for three pages, you need three free buffers for that page size.
 - Gets the required resource manager and **reads the required pages in the free buffers** (for this operation, use the method `readPagesFromResource(byte[][] buffers, int firstPageNumber)`, which is provided by the resource manager).
 - Then add and pin every requested page (multiple pins may be required!)
 - Finally, notifies all interesting threads that the request is completed
-  HINT: Use the `LoadQueueEntry` as the monitor! Similar to the example in the previous slide → The execution thread waits for the notification of the read thread

Threads in Exercise 3

- You need to implement a WriteThread
 - Checks if the write queue is not empty.
 - Gets the next write request and processes it:
 - Gets the required resource manager and writes the new value of the page back to disk (for this operation, use the method `writePagesToResource(byte[][] buffers, CacheableData[] wrappers)`, which is provided by the resource manager).
 - Finally, return the buffers to the BufferPoolManager, by adding them to the free buffer list.

WARNING! If the BufferPoolManager has been closed, all threads must be stopped, but first, the write thread must write all values back to disk! Otherwise, we will lose changes, and the database will be in an inconsistent state.

Monitors in Exercise 3

- Use the LoadQueueEntry as the monitor for load requests.
- Don't forget to block access to the data structures to avoid access conflicts and race conditions
 - This applies to the list of free buffers, the caches, and the request queues.
 - Don't forget there are multiple cache objects (one for each PageSize).
 - Use **a single monitor object** to block the load and write queues simultaneously.

BufferPoolManager API

BufferPoolManager API

- Initializing the BufferPoolManager
- Create a new page
- Prefetch pages
- Get a page
- Unpin pages
- Closing the BufferPoolManager

Initializing the BufferPoolManager

- `void startIOThreads()`
 - Initializes the I/O threads (read and write thread).
 - Then, starts the operation of the I/O threads in the BufferPoolManager.
- `void registerResource(int id, ResourceManager manager)`
 - Registers a resource with this BufferPoolManager.
 - Don't forget to initialize a list of free buffers and a cache of the corresponding PageSize, if the resource has a PageSize that hasn't been registered before.
 - You can find the size parameters of the buffer list and cache in the Config object that has been passed in the constructor.

Create a New Page

- `CacheableData createNewPageAndPin(int resourceId)`
 - Creates a new empty page for the resource described by the id and pins the new page.
 - You must take a free buffer of the corresponding `PageSize` and call the `ResourceManager#reserveNewPage(byte[])` to create a new page inside that buffer.
- `CacheableData createNewPageAndPin(int resourceId, Enum<?> type)`
 - Analog to the last one. The only difference is that it lets the caller specify the type of the page to be created. That is important for example, for indexes, which have multiple types of pages (leaf pages, inner node pages).
 - For you, the only difference that matters is that you need to use the method `ResourceManager#reserveNewPage(byte[], Enum<?> type)` to create the page. Otherwise, is exactly the same as the previous method.

Prefetch Pages

- `void prefetchPage(int resourceId, int pageNumber)`
 - Prefetches a page. If the page is currently in the cache, this method causes a hit on the page.
 - If not, a request to load the page is issued. When the read thread has loaded the page, it adds it to the cache **without causing it to be considered hit.**
- `void prefetchPages(int resourceId, int startPageNumber, int endPageNumber)`
 - Prefetches a sequence of pages and behaves exactly like the previous method, only that it prefetches multiple pages.

Get a Page

- `CacheableData getPageAndPin(int resourceId, int pageNumber)`
 - Transparently fetches the page defined by the page number for the given resource.
 - If the cache contains the page, it will be fetched from there; otherwise, it will be loaded from secondary storage.
 - The page is pinned in order to prevent eviction from the cache.

Unpin Pages

- `void unpinPage(int resourceId, int pageNumber)`
 - Unpins a page so that it can again be evicted from the cache.
 - It will try to unpin the page but will not throw any exception if the page is not contained in the cache or if it is not pinned.
- `CacheableData unpinAndGetPageAndPin(int resourceId, int unpinPageNumber, int getPageNumber)`
 - Unpins a given page and in addition fetches another page from the same resource.
 - This method works exactly like the method `getPageAndPin(int resourceId, int pageNumber)`, only that it unpins a page before beginning the request for the new page.

Exercise 3

Exercise 3

- Implement the BufferPoolManager interface.
- Implement the factory method createBufferPoolManager.

Tests and points

Test name	Points	Description
testClosing	2,5	Tests whether the BufferPoolException is correctly thrown after the BufferPoolManager has been closed.
testCreateNewPageAndPin	2,5	Tests the creation of pages.
testGetPageAndPin	2,5	Tests the retrieval and pinning of tuples through the BufferPoolManager.
testPrefetchPage	2,5	Prefetch pages from the BufferPoolManager and test that returning from it is faster than the wait time in the mock resource manager.
testSynchronization	4	Tests whether the synchronization works without deadlocks.

